Chemical Computing for Distributed Systems: Algorithms and Implementation

Marko Obrovac

▶ To cite this version:

Marko Obrovac. Chemical Computing for Distributed Systems: Algorithms and Implementation. Calcul parallèle, distribué et partagé [cs.DC]. Université Rennes 1, 2013. Français. NNT : . tel-00925257

HAL Id: tel-00925257
https://theses.hal.science/tel-00925257v1

Submitted on 7 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE RENNES 1 ueb

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Marko OBROVAC

préparée à l'unité de recherche INRIA / IRISA – UMR6074
Institut de Recherche en Informatique et Système
Aléatoires ISTIC

Chemical Computing for Distributed Systems : Algorithms and Implementation

Thèse soutenue à Rennes le 28 Mars, 2013

devant le jury composé de :

Jean-Pierre BANÂTRE
Professeur Emerite, Université de Rennes 1 / *Président*

Pierre SENS
Professeur, Université Pierre et Marie Curie / *Rapporteur*

Jean-Louis GIAVITTO
Directeur de Recherche, IRCAM / *Rapporteur*

Manish PARASHAR
Professor, Rutgers, New Jersey / *Examinateur*

Thierry PRIOL
Directeur de Recherche, INRIA Rennes / *Directeur de thèse*

Cédric TEDESCHI
Maître de Conférences, Université de Rennes 1 / *Co-directeur de thèse*

Abstract

With the emergence of highly heterogeneous, dynamic and large distributed platforms, the need for a way to efficiently program and manage them has arisen. The concept of autonomic computing proposes to create self-manageable systems — systems which are aware of their components and their environment, and can configure, optimise, heal and protect themselves. In the context of realisation of such systems, declarative programming, whose goal is to ease the programmer's task by separating the control from the logic of a computation, has regained a lot of interest recently. In particular, rule-based programming is regarded as a promising model in this quest for adequate programming abstractions for these platforms. However, while these models are gaining a lot of attention, there is a demand for generic tools able to run such models at large scale.

The chemical programming model, which was designed following the chemical metaphor, is a higher-order, rule-based programming model, with a non-deterministic execution model, where rules are applied concurrently on a multiset of data. In this thesis, we propose the design, development, and experimentation of a distributed chemical runtime for generic, largely-distributed platforms.

The architecture proposed combines a peer-to-peer communication layer with an adaptive protocol for atomically capturing objects on which rules should be applied, and an efficient termination-detection scheme. We describe the software prototype implementing this architecture. Based on its deployment over a real-world test-bed, we present its performance results, which confirm analytically obtained complexities, and experimentally show the sustainability of such a programming model.

Keywords :
distributed systems, structured peer-to-peer networks, chemical programming model, distributed runtime

Résumé

Avec l'émergence de plates-formes distribuées très hétérogènes, dynamiques et à large-échelle, la nécessité d'un moyen de les programmer efficacement et de les gérer a surgi. Le concept de l'informatique autonomique propose de créer des systèmes auto-gérables — des systèmes qui sont conscients de leurs composants et de leur environnement, et peuvent se configurer, s'optimiser, se guérir et se protéger. Dans le cadre de la réalisation de tels systèmes, la programmation déclarative, dont l'objectif est de faciliter la tâche du programmeur en séparant le contrôle de la logique du calcul, a retrouvé beaucoup d'intérêt ce dernier temps. En particulier, la programmation á base de des règles

est considérée comme un modèle prometteur dans cette quête pour des abstractions de programmation adéquates pour ces plates-formes. Cependant, bien que ces modèles gagnent beaucoup d'attention, ils créent une demande pour des outils génériques capables de les exécuter à large échelle.

Le modèle de programmation chimique, qui a été conçu suite à la métaphore chimique, est un modéle de programmation à bas de règles et d'ordre supérieur, avec une exécution non-déterministe modèle, où les règles sont appliquées simultanément sur un multi-ensemble de données. Dans cette thèse, nous proposons la conception, le développement et l'expérimentation d'un intergiciel distribué pour l'exécution de programmes chimique sur des plates-formes à large échelle et génériques.

L'architecture proposée combine une couche de communication pair-à-pair avec un protocole de capture atomique d'objets sur lesquels les règles doivent être appliquées, et un système efficace de détection de terminaison. Nous décrivons le prototype d'intergiciel mettant en oeuvre cette architecture. Basé sur son déploiement dans un banc d'essai réel, nous présentons les résultats de performance, qui confirment les complexités analytiques obtenues et montrons exérimentalement la viabilité d'un tel modèle de programmation.

Mots-clés :
systèmes répartis, réseaux pair-à-pair structurés, modèle de programmation chimique

Mr. Frankle began to suffer from a very serious disease — the computer disease. The disease with computers is you play *with them.*

– Richard P. Feynman

Contents

Introduction 1

I Preliminaries 17

1 Background 19
1.1 Chemical Programming Model . . . 20
1.1.1 GAMMA . . . 20
1.1.2 Higher-order Chemical Language . . . 22
1.1.3 Inertia Detection . . . 27
1.2 Distributed Hash Tables . . . 28
1.2.1 Overview . . . 29
1.2.2 Some DHTs . . . 30
1.2.3 Range Queries . . . 36
1.3 Mutual Exclusion . . . 39
1.3.1 Single-resource Mutual Exclusion . . . 40
1.3.2 Multiple-resource Mutual Exclusion . . . 41
1.3.3 k-out of-M-Mutual Exclusion . . . 43

2 Physical Parallelism 47
2.1 Single-processor Execution . . . 48
2.2 Message-passing Methods . . . 49
2.2.1 Centralised Controller . . . 50
2.2.2 Moving Values . . . 52
2.2.3 Odd-even Transposition . . . 53
2.2.4 Fold-over Operation . . . 54
2.3 Shared-memory Approach . . . 56
2.3.1 Parallel Implementation . . . 56

2.3.2 Inertia Detection . . . 57
2.3.3 Experiments . . . 58
2.4 Conclusion . . . 58

II Distributed Chemical Computing 61

3 Feasibility Study 63
3.1 DSM-based Execution Platform . . . 64
3.1.1 DSM-inspired Architecture Overview . . . 65
3.1.2 Course of Execution . . . 66
3.1.3 Issues of the DSM-based Platform . . . 67
3.2 Hierarchical Execution Platform . . . 68
3.2.1 Physical Layer Abstraction . . . 69
3.2.2 Execution Flow . . . 69
3.2.3 Condition Checking and Inertia Detection . . . 71
3.2.4 Tree Reorganisation . . . 77
3.3 Prototype . . . 80
3.4 Evaluation . . . 81
3.4.1 Test Programs . . . 81
3.4.2 Results . . . 82
3.5 Conclusion . . . 88

4 Atomic Capture of Multiple Molecules 91
4.1 System Model . . . 93
4.2 Protocol for the Atomic Capture . . . 94
4.2.1 Pessimistic Sub-protocol . . . 95
4.2.2 Optimistic Sub-protocol . . . 98
4.2.3 Sub-protocol Mixing . . . 98
4.2.4 Dormant Nodes . . . 100
4.3 Execution of Multiple Rules . . . 101
4.3.1 Multiple Success Rates . . . 102
4.3.2 Initial Rule Assignment . . . 102
4.3.3 Changing the Active Rule . . . 102
4.3.4 Discussion . . . 103
4.4 Proof of Correctness . . . 104
4.4.1 Proof of Safety . . . 104
4.4.2 Liveness Proof . . . 104
4.4.3 Convergence Time . . . 106
4.5 Evaluation Set-up . . . 106
4.6 Experiments Involving One Rule . . . 107
4.6.1 Execution Time . . . 107
4.6.2 Switch Threshold Impact . . . 108
4.6.3 Switch Behaviour . . . 109

4.6.4 Communication Costs . . . 111
4.7 Experiments with Multiple Rules . . . 113
4.7.1 Multiple-rule Test Programs . . . 113
4.7.2 Execution of the Independent-rules Program . . . 114
4.7.3 Execution of the Dependent-rules Program . . . 116
4.7.4 Execution of the Circular Program . . . 118
4.7.5 Execution of the Workflow Program . . . 118
4.8 Conclusion . . . 122

5 Decentralised Execution Platform **123**
5.1 Platform Overview . . . 124
5.1.1 Initialisation . . . 125
5.1.2 Execution . . . 125
5.1.3 Termination . . . 127
5.2 Data Structures and Algorithms . . . 127
5.2.1 Double DHT Layer . . . 127
5.2.2 Random Meta-Molecule Fetch . . . 129
5.2.3 Search for Candidates . . . 130
5.2.4 Atomic Grab of Molecules . . . 130
5.2.5 Complexity Analysis . . . 131
5.3 Execution of Higher-order Programs . . . 132
5.3.1 Execution of Rules . . . 133
5.3.2 Correctness of Execution . . . 134
5.3.3 Inertia Detection . . . 136
5.4 Software Prototype . . . 137
5.4.1 Entities . . . 137
5.4.2 Execution Cycle . . . 138
5.4.3 Optimisations . . . 140
5.5 Evaluation . . . 140
5.5.1 Test Programs . . . 140
5.5.2 Experimental Results . . . 142
5.6 Conclusion . . . 149

III Conclusions And Addenda **151**

Conclusion **153**

Bibliography **168**

List of Publications **169**

A Résumé en Français **171**
A.1 Préliminaires . . . 173
A.1.1 Modèle de programmation chimique . . . 173
A.1.2 Modèle du Système . . . 174
A.2 Protocole Pour la Capture Atomique de Molécules . . . 175
A.2.1 Sous-protocole Pessimiste . . . 175
A.2.2 Sous-protocole Optimiste . . . 176
A.2.3 Cohabitation des Sous-protocoles . . . 179
A.2.4 Sureté et Vivacité du Système . . . 179
A.3 Evaluation . . . 181
A.4 Travaux Connexes . . . 183
A.5 Conclusion . . . 184

Introduction

One of the initial reasons for the birth of distributed systems was the scientists' need to extend the computing power of existing parallel systems. At first, those were homogeneous, tightly-coupled systems confined to physical closeness, called clusters. As this research field grew, scientists started interconnecting clusters over wide areas, bringing about *grids*. On the other hand, the rapid increase in number and computing power of commodity hardware enabled the emergence of so-called *desktop grid* systems: the general public joined in on projects such as SETI@Home[1] or Folding@Home[2]. The numerous improvements made to both scientific and desktop grids, coupled with the rapid developments in the area of high-speed networking, paved the way to the widespread adoption of distributed systems in the form of *cloud computing*. This model of distributed computing offers *elasticity* to its users, since they do not have to own the system, hardware nor software, they plan to utilise for their computations. This form of *on-demand computing*, realised through a *pay-as-you-go* business model, enables one to use only the resources actually needed for a given computation: on the hardware level (infrastructure-as-a-service, IaaS), on the platform or middleware level (platform-as-a-service, PaaS), or on the application level (software-as-a-service, SaaS).

At the same time, due to technological progress and advancements, a variety of mobile devices able to connect to different network have come about, ranging from network disks and printers to consumer mobile phones, television sets and other intelligent appliances. Nowadays, a significant part of devices directly accessing Internet are small mobile devices.

Intertwining these two worlds results in complex, heterogeneous systems of unprecedented scale, ever so more intertwined with our everyday lives. To grasp the impact these systems can have, consider that, when in April 2011 user web servers and their sites hosted at Amazon's Elastic Compute Cloud (EC2) were unavailable during three days, they impeded their customers to function, causing important losses [1]. While the failure was

[1] http://setiathome.ssl.berkeley.edu/
[2] http://folding.stanford.edu/

physical, the user experience of the outage occurred as a direct consequence of relying on centralised architectures: when a central entity fails, whole (sub-)systems become unavailable in spite of the fact that they are themselves operational.

Therefore, when faced with systems of such scale, decentralised architectures comprised of self-contained entities become a necessity, allowing participants to communicate and exchange information and/or computation regardless of the state or actions of others. Interestingly, the repercussions of adopting such a research direction might have got reach well outside the domain of distributed systems, both in practical and theoretic terms. On the practical side, consider that most electricity-distribution networks use a centralised, hierarchical organisational scheme, which could potentially lead to massive power outages [2]. In this case, proposing a decentralised distribution network might localise the consequences of hardware malfunction. As for the theoretical side, in recent years a new research field has opened up, that of studying complex systems found in nature [16]. Since nature represents the ultimate large-scale, heterogeneous and most complex decentralised system there is, research in the field of distributed systems can yield constituents used in modelling biological systems. An example of this is the recently-proposed first whole-cell computational model of a live organism [65]. Each of the 28 modules mimics one of the one-cell bacterium's function, such as *RNA* to *DNA* transcription, protein formation and photosynthesis. As each of them communicates only with a few other modules throughout its functioning cycle, they are coordinated in a decentralised way.

Adopting a decentralised, self-contained view of currently available distributed platforms in order to leverage their capabilities calls for new ways to conceive, model and programme them. *Autonomic computing* [66] is a paradigm consisting in building computing systems that can "*manage themselves in accordance with high-level guidance from humans*" [96]. It has been proposed analogously to the autonomic nervous system which takes care of unconscious reflexes in the human body, such as digestive functions, respiration or constriction of blood vessels. To fulfil the requirement of self-management, an autonomic computing system must have, like its biological counterpart, a mechanism whereby changes in its essential variables can trigger changes in the behaviour of the system, using appropriate solutions based on current state, context and content and on specified policies. Namely, to minimise (and later completely remove) human intervention, an autonomic system must have the following properties [55], also referred to as *self-* properties*:

- *self-awareness*: an autonomic system must have detailed knowledge of its components, current status, ultimate capacity and all connections with other systems in order to govern itself, *i.e.* it is able to "know itself" and is aware of its behaviours so as to be able to manage itself *via* the self-* properties described below;

- *self-configuration*: an autonomic system configures itself according to high-level goals which specify what is desired, but not necessarily how to accomplish it;

- *self-optimisation*: an autonomic system *proactively* optimises its use of resources in order to improve performance and/or quality of service;

- *self-healing*: an autonomic system detects, diagnoses and, to the extent possible, fixes problems, both low-level, hardware and high-level, software ones;
- *self-protection*: an autonomic system protects itself from malicious attacks and careless users, as well as tunes itself to achieve security, privacy and data protection in such a way as to, ideally, anticipate security breaches and prevent them from occurring in the first place.

These properties define the behaviours and inner workings of an autonomic system which allow it to satisfy the given constraints and goals. However, in order to accomplish them, and consequently minimise human intervention, it must communicate with its environment and adapt to it accordingly. Hence, three more properties are imposed upon it:

- *context awareness*: an autonomic system is aware of its execution environment and the context surrounding its activity and is able to react to changes in the environment through processes such as the tapping of available resources or the negotiation of usage of its elements by other systems, and in this way changing both itself and its environment;
- *openness*: an autonomic system implements and is built upon open standards, since it must function in a heterogeneous world and be portable across multiple hardware and software architectures;
- *anticipation*: an autonomic system is able to anticipate, to the extent possible, its needs and actions and those of its context and manage itself proactively.

One of the areas where the strengths of autonomic computing could be fully leveraged is the emerging field of *Internet of Services* (IoS) [23]. A service can be defined as a self-contained software utility that exposes a capability or information as a reusable unit. Examples of these include currency exchange services, chat services, data storage services, etc... Placing them in the context of Internet enables their *reusability*: multiple users can use the same service for their own, different purposes. Consequently, an increasing number of applications are created by *composing* and *combining* services in a temporal manner, making these systems highly dynamic and loosely coupled. These applications, such as BLAST[3], CardiacAnalysis[4] or Montage[5], can easily be shared amongst people in different communities, as shows the example of the *myExperiment*[6] platform.

Since a particular service implementation can be used in multiple applications, strengthening it with self-management capabilities offered by autonomic computing appears to be a prerequisite for its reliable functioning. In this way, services would be able to: *(i)* configure themselves, in case their environment changes due to a system-wide update or relocation; *(ii)* optimise their computations based on the resources′ loads and on

[3] *http://www.blastalgorithm.com/*
[4] *http://www.creatis.insa-lyon.fr/site/*
[5] *http://montage.ipac.caltech.edu*
[6] *http://www.myexperiment.org*

the level of current user demand — more users means more resource utilisation; *(iii)* protect themselves from malicious users and unwanted behaviours; and *(iv)* heal themselves when, *e.g.*, a computational error is encountered. Moreover, the connections between services themselves could be made *autonomic*. Alternative routes from one service to the other could be provided in case a connection drops and transfer of data might be optimised to use faster or multiple routes, all the while protecting the connection from intrusions and keeping it secure. On top of such a robust architecture, applications would be fairly easy to compose, allowing the user to focus only on application-specific aspects.

Motivation

In spite of the fact that the concept of autonomic computing has been around for a decade now, its vision is far from being fulfilled [37]. Due to the scale, dynamics and heterogeneity of today's emerging platforms, such as the aforementioned Internet of Services, fully leveraging their computing power remains a widely open issue. In particular, abstracting out the technical details of the low-level machinery of the platform appears to be a prerequisite to actually being able to efficiently compute over it. In other words, the logic of the computation (which does not change, regardless of the underlying platform characteristics) should be separated from its low-level implementation.

This situation advocates the use of declarative programming [76], whose goal is to separate the logic of a computation (*"what we want to do"*) from its control (*"how to achieve it"*). More precisely, while the "what" is to be defined by the programmer, the "how" becomes for them implicit, hidden inside the system.

In particular, rule-based programming, where this logic is expressed as a set of high-level *rules* allows to hide the intrinsic difficulties of the parallelism and distribution of the runtime from the programmer. Rule-based programming provides a common framework in which computation can be seen as a set of logical statements about how a system operates. The majority of systems based on it use the so-called *Event-Condition-Action* (ECA) type of rules, which was originally proposed as a formalism for active capabilities in database management systems [84]. When using this formalism, the programmer specifies *(i)* a triggering event; *(ii)* a list of conditions which have to be checked; and *(iii)* the action to perform. When an event corresponding to one of the rules' definitions occurs in the system, the rule is *activated*. Its list of conditions is checked and, in case all of the listed conditions have been met, the action is performed. Additionally, the programmer can specify an optional post-condition confirming the state change after the execution of the action. Rules define the flow of the execution and its constraints, as well as control the behaviour of the system by *reacting* to events. Composing multiple ECA rules allows one to construct *event-driven applications*. There is no sequentiality in such application since the execution is defined through the activation of rules, implying that there is no need to order or schedule the execution. Furthermore, as the level of abstraction of these rules is high, the programmer does not have to worry about the implementation's details.

Recently, some work has gone into showing how to concretely apply rule-based pro-

gramming to the specification of distributed systems. For instance, in [51], it has been shown how communication protocols and peer-to-peer applications can be specified using a rule-based language. In [3], the same programming style is applied to web-based data management. On the computing side, rule-based programming was also used as a building block for workflow management systems [69, 124].

Example

Let us illustrate this declarative, rule-based programming style on a generic example of a service-based platform, where users submit their *workflows*, *i.e.*, temporal compositions of services, to be executed within the platform.

Notation. Throughout the example we are going to use rules of the form:

rule *rule_name* = **if** *condition* **then** *action*

The rule *rule_name* is going to be triggered once the condition *condition* becomes true, and the action *action* is going to be performed. Conditions are usually simple expressions which test for the presence of a certain object, or a set thereof. Actions involve procedure calls (of the form *X(...)*) and insertions or removals of objects (using the keywords **insert** and **remove**, respectively). Note that, if there are multiple actions to be performed (concatenated with the **and** keyword), they are going to be executed in parallel.

Workflows. A workflow is a set of tasks to be executed following an order defined by some data and control flows. Consider the simple workflow pictured on the left side of Figure 1. The workflow comprises four services, linked through some data dependencies. Such a workflow can be expressed through the two rules on the right side of the figure, where the expression $S_i(\cdots)$ denotes the call to the actual service S_i, while Res_j expresses (the presence of) the result (or output) of service S_j. The existence of this object triggers the actual execution of the service and insertion of the result. The first rule expresses the *fork* (production of the result of S_1 triggers the execution of both S_2 and S_3) while the second one expresses the *join*.

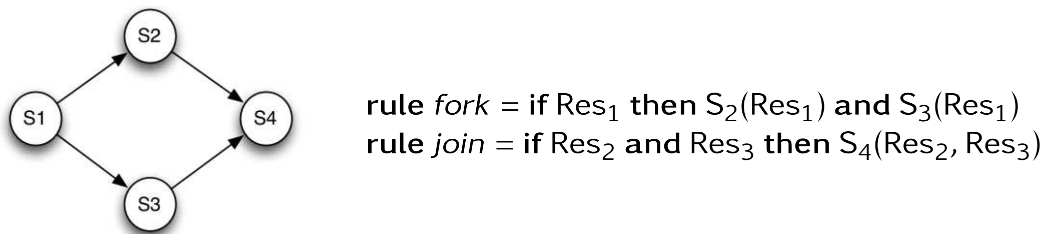


Figure 1: A simple workflow and its rule-based expression.

Sometimes, due to some failures, some task may not be able to produce its expected results. One way to tackle this problem at the workflow level is to be able to modify it on the fly, upon the detection of the failure.

For the sake of illustration, let us consider a computation on a matrix decomposed in blocks where each block is to be processed independently in parallel. Given an initial task separating the matrix into blocks and triggering all of the corresponding tasks, the workflow starts as pictured in Figure 2 (left). This *fork* pattern can be expressed by the following rule, Res_0 being a pointer table in which each task $T_1, \cdots, T_n$ is able to collect the information about the block of the matrix assigned to it (assuming the matrix itself is not transmitted, but stored elsewhere, *e.g.* in a memory space shared between the tasks):

$$\textbf{rule}\ \mathit{matrix_fork} = \textbf{if}\ \mathrm{Res}_0\ \textbf{then}\ \mathrm{T}_1(\mathrm{Res}_0[1])\ \textbf{and}\ \mathrm{T}_2(\mathrm{Res}_0[2])\ \textbf{and} \ldots\ \textbf{and}\ \mathrm{T}_n(\mathrm{Res}_0[n])$$

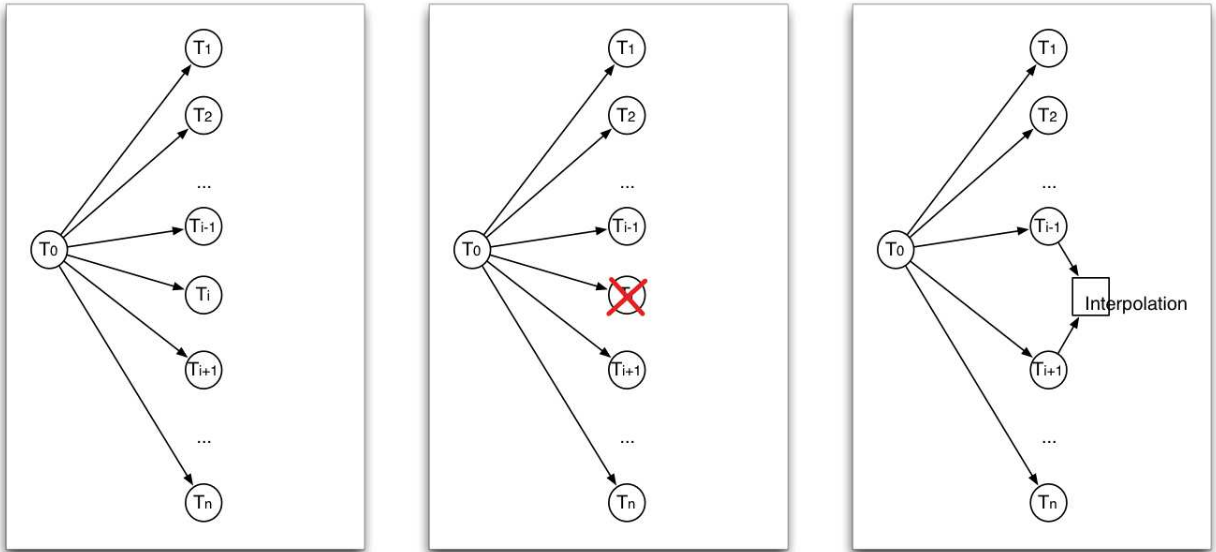


Figure 2: Matrix processing workflow: initial state, failure, interpolation.

Let us now assume that the processing of some block encountered a failure, as illustrated on Figure 2 (middle). One common technique for avoiding the recalculation of the missing result is using the successfully computed results in the neighbourhood of the problematic block to infer the missing result by interpolation. This leads to the modification of the workflow itself : the deletion of the failed task, and the creation of a new task performing the interpolation, pictured as a square in Figure 2 (right). Note that the interpolation creates new data dependencies within the workflow. It means that we need to be able to change rules dynamically. Let us consider the new rule modelling the interpolation task along with its data dependencies, parametrised with the index i of the task.

$$\textbf{rule}\ \mathit{interpolate}_i = \textbf{if}\ \mathrm{Res}_{i-1}\ \textbf{and}\ \mathrm{Res}_{i+1}\ \textbf{then}\ \mathrm{Interpolation}(\mathrm{Res}_{i-1}, \mathrm{Res}_{i+1})$$

This new rule has to be injected in the program dynamically to materialise the change in the workflow, which can be achieved through the higher order. Higher-order programming models regard functional parts of a program (here, rules) as *regular* objects. The programmer is, thus, allowed to manipulate a rule through another rule, and in doing so modifying the course of the execution on the fly. The injection of the *interpolate* rule is done by the following (higher-order) rule.

rule $inject_interpolation_i$ = **if** T_i.status = Failed **then insert** $interpolate_i$

When the task T_i fails, its status is set to "Failed". This is detected by the *inject_interpolation* rule, which introduces the *interpolate* rule, effectively triggering the interpolation procedure. Upon completion, it delivers the result in the form of a Res_i object, just as the original task would, in this way allowing the computation to continue smoothly. It is important to notice here that the $inject_interpolation_i$ rule is present from the beginning of the execution, while $interpolate_i$ is dynamically inserted only after a failure has been detected.

Services. Let us now concentrate on the building block of the workflow, namely, the task, which was until now considered as a black box, the actual service called being abstracted out. In other words, we only considered *abstract* workflows. In the following, we focus on how every task taking part of the computation can be made autonomic — providing self-optimisation and self-healing — through rule-based programming. A possible rule-based specification of such a task is illustrated in Figure 3. The big circle is the *context* of the autonomic element — a task —, and includes all of the data and rules needed. The two rectangles represent two underlying processes injecting some information into the task's context. More specifically, the *service discoverer* gathers information about the currently available services implementing this task and their performances and to inject it through objects of the form S_i = (free_cpu$_i$, net_tput$_i$) including a list of criteria (here CPU and network throughput) that could be arbitrarily extended. The *failure detector* is in charge of detecting the failure of the currently bounded service, and injecting this failure detection in the form of a specific object. Henceforth, we assume this object to be symbolised by the string "failure_detected".

As several implementations for a given task may coexist in the platform, the objective behind self-healing is, upon detection of the failure of the currently bounded service (represented by the *Binding* object), to autonomously switch to another service, among those injected into the rule-based engine by the service discoverer. This is achieved by the *repair* rule:

rule *repair* = **if** "$failure_detected$" **then** Binding = S_j **and remove** S_i

The objective behind self-optimisation is to bind the service to the best one currently available, according to some predefined metric. This objective is achieved by using the *optimise_cpu* rule, which constantly tries to select the best implementation, according to the CPU criterion:

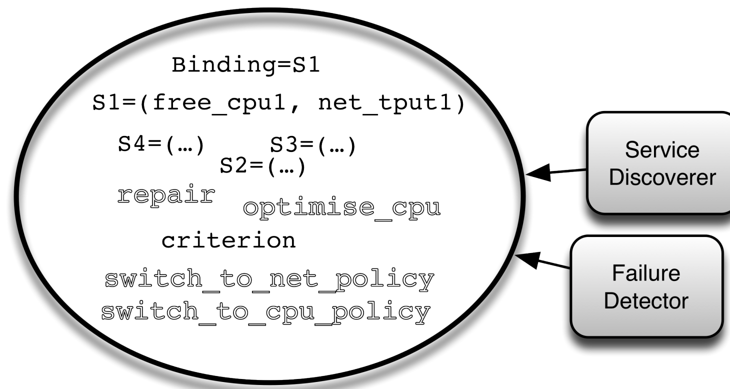


Figure 3: Rule-based specification of an autonomic service.

rule *optimise_cpu* = **if** Binding = S_i **and** free_cpu$_j$ > free_cpu$_i$ **then** Binding = S_j

Similarly, an *optimise_net* rule could consider a service's network capabilities. Even though the possibility of having different policies brings more flexibility to the adaptation, it creates the need for dynamic switching from one to the other based on a criterion, in our case meaning *optimise_cpu* might need to be put aside in favour of *optimise_net*. This is again possible through the higher order and the *criterion* object, using the *switch_to_net_policy* rule:

rule *switch_to_net_policy* = **if** criterion = "Net" **then**
insert *optimise_net* **and remove** *optimise_cpu*

Furthermore, as illustrated on Figure 3, other rules, for instance *switch_to_cpu_policy*, can be similarly constructed and introduced in the context concurrently. These rules can coexist smoothly, as the criterion can take only one value at a time, preventing concurrent reactions of contradictory switching rules. Note that, even though all of the services use the same rules to guide their executions, due to the *isolation* of each service, achieved through the usage of different contexts, every one of them is able to bind services based on different criteria and policies independently and autonomously.

Summary. The example presented is a two-tier architecture for enacting, executing and dynamically managing workflows at run time. The user supplies exclusively the definition of the workflow to execute — the *"what to do"* — and is freed from any consideration of implementation. On the other side, the system is in charge of finding the appropriate bindings for tasks, connecting them through the user-defined control and/or data dependencies, and monitoring the execution itself in an autonomic fashion — the *"how to do it"*.

It does so on two levels. On the lower, task level, each service binding behaves as a *managed autonomic element, i.e.* each of them is handled independently from the others

as a stand-alone sub-system; different services are monitored for performance according to different metrics and individual service failures are handled locally. On the higher, workflow level, the system manages a workflow in its entirety through the control of the flow of the execution according to the data and control dependencies defined. Moreover, if an individual task fails to complete, recovery procedures are introduced in order to ensure the continuation of the computation.

Concurrency and Rule-based Programming: Chemical Programming Model

As we have showed, a rule-based model is able to provide the appropriate abstractions for both users and administrators of high-level systems, such as the one discussed in the above example. Still, this example shows that the particular model chosen must possess two crucial properties:

- *Concurrency*. Multiple rules coexist in a program and they can be activated and applied concurrently. Therefore, the programmer must be able to express parallel actions in an easy and intuitive way. On the other hand, the chosen model's runtime must be able to support such parallel executions.

- *Higher order*. Due to the dynamic nature of the underlying platform, changes and deviations in a program's flow have to be considered as a regularity and thus communicated effortlessly by the programmer. As shown in the example above, altering the execution's flow is not only a desirable feature, but a much-needed one in the context of autonomic computing. Without it, the design and implementation of such a self-managed system would require complex execution flows to be devised, increasing the possibility of error. Hence, the programming model chosen has to be equipped with higher-order capabilities. They allow one to regard, and use, functional parts of a program as ordinary data (namely, variables), enabling the programmer to instruct the runtime to change the flow of the execution in a clean and clear manner.

Consequently, in this thesis we focus on the chemical programming model, which associates rule-based programming with an implicitly-parallel runtime. It also integrates the higher order, *i.e.* the ability to manipulate rules as ordinary objects, and thus to modify the program dynamically at run time. Initially proposed as the GAMMA formalism [11], the chemical metaphor envisions the execution of a program as a set of reactions between molecules moving and colliding autonomously according to Brownian motion[7]. On collisions, and according to some pre-defined rules (constituting the chemical program), molecules are consumed and new molecules (and concomitantly new information) are

[7] Brownian motion is the macroscopic picture emerging from a particle moving randomly in d-dimensional space. On the microscopic level, at any time step, the particle receives a random displacement, caused by other particles hitting it or by an external force. The movement of the particle can be, hence, modelled by a random walk.

created. Reactions take place in an implicitly-parallel and distributed fashion, and independently from each other. Therefore, the model allows one to concentrate on the logic of the computation at design time, and to write programs cleared of any artificial sequentiality, arbitrary scheduling or synchronisation brought by implementation's considerations.

Throughout the years, the expressiveness of the chemical model has been leveraged in different contexts, for instance to verify shared-memory coherence protocols [86], to process large images [85], or to specify software architectures [57, 73]. The model has recently been raised to the higher order, giving birth to the Higher-Order Chemical Language (HOCL) [13]: in HOCL, everything is a molecule, including reaction rules, that can themselves be consumed or produced upon a reaction. This feature makes the model highly adequate in today's emerging platforms, especially in the context of large-scale coordination of entities [14, 41, 89, 34, 93].

Problem Statement

While the chemical programming model has been shown to have the suitable level of abstractions needed to construct autonomic systems, the problematic of the actual execution of chemical programs has been mostly left aside in the literature. In order to fill the existing gap between the existing conceptual works and their quite limited practical implementations, the following characteristics must be taken into account:

Heterogeneity. The programmer using the chemical specification should be concerned only with the program's logic; it is up to the runtime to ensure the program's correct execution regardless of the underlying hardware's and software's specific constraints. Therefore, the chemical platform must run in diverse, heterogeneous environments.

Scalability. Due to their size and the vast number of components and entities, the design of large-scale systems should be thought through the paradigm of autonomic computing, since it aims at alleviating humans from maintenance and management of computing systems. The chemical platform must, thus, employ scalable protocols and algorithms for the execution of programs in order to face the challenge of scale.

Decentralised Execution. The entities participating in an autonomic system are spread around these large-scale and heterogeneous environments. Moreover, their communication patterns and interactions are not known *a priori*. In such conditions, it is crucial for the chemical platform to be able to execute programs in a decentralised way as a centralised solution might easily encounter bottleneck problems.

The goal of this thesis is to provide a generic decentralised platform dedicated to the execution of higher-order chemical programs. We envision a high number of nodes willing to collaborate, with each collaborating node equipped with an engine executing rules with the molecules available in the system, but not necessarily present on the node executing

the rule. Keeping in mind the aforementioned requirements, the following four issues have to be tackled in order to fulfil this objective:

1. **Communication Abstraction**. Each node has to be able to communicate with every other node and it has to do so in an efficient way and over heterogeneous underlying environments.

2. **Molecule Discovery**. Molecules are dispatched over the network, meaning suitable reaction candidates have to be found efficiently in spite of the scale of the platform.

3. **Atomic Capture**. Once the appropriate molecules have been located, a node must grab *all of them* atomically, as other nodes may try to fetch them as well at the same time. In other words, situations where multiple nodes try to obtain the same molecule have to be resolved in favour of only one of them, seeing that a molecule can react at most once. Performing more than one reaction with the same molecule could brake the logic of the program. Mutual exclusion, therefore, has to be enforced during this process.

4. **Detection of Termination**. To secure the termination of a program, we need to ensure to detect the fact that no more reactions are possible. This detection, when done in a centralised way, has a combinatorial complexity. This suggests that relying on intelligent information retrieval techniques is mandatory in order to circumvent the problem.

For the purpose of clarification, consider the following example. Let us suppose a network of nodes (of unknown size) has to execute the following program, written using the previously-described formalism:

rule *sum* = **if** $x \neq y$ **then insert** $x + y$ **and remove** x, y

on the following data set:

$2, 3, \ldots, 1000$

The rule *sum* is activated when two integers with different values are found. Their sum is stored into the shared space, while the integers themselves are removed from it. Reapplying this rule yields the sum of all of the integers in the set. Note that, while the concrete problem presented here is easily soluble using only one mathematical formula computed on a single machine, the purpose of the example is to show the different issues encountered when executing a rule-based (chemical) program in a large-scale environment.

As noted earlier, the number of nodes participating in the execution is not known. Moreover, the chemical processes running on these nodes might be executed in different environments. Therefore, the communication layer must allow nodes to communicate efficiently and over heterogeneous underlying environments (Point 1 above). Suppose the integers are spread in the network, *e.g.* 2, 3 and 4 reside on node N_1, 5 and 6 on node

N_2, and so on. Nodes have to be able to find them in an efficient manner. A node might look the integers up locally, and then, if candidates cannot be found, it performs a search in the network to locate integers satisfying the condition. In this example, any two integers from the data set can be paired up, but rules usually have more restrictive conditions. Thus, discovering potential candidates (Point 2 above) is essential to the progress of the computation. The next step is to fetch the candidate objects found in an atomic fashion: all of them have to be captured in order to ensure the rule is going to be applied. Suppose node N_1 decides to grab the integers 2 and 3, while node N_2 wants to fetch 3 and 4. Both nodes would have to capture both of their candidates in order to apply the *sum* rule. However if they both succeeded, the execution's correctness would be compromised, breaking the program's logic: N_1 would produce 5 and N_2 7, which would be incorrect since the sum of 2, 3 and 4 is 9 and not 12. Therefore, mutual exclusion must be enforced to guarantee the correctness and consistency of the execution (Point 3 above). These steps are repeated while there are candidates to apply the rule onto. In the example, each time a node performs a sum of two integers, the size of the data set decreases until there is only one integer left — the overall sum of the data —, signalling the end of the execution. All of the nodes have to detect the program's termination in an efficient manner in order to minimise the execution time and network traffic generated (Point 4 above).

Contribution

Central to this thesis is the premise that the construction of a *distributed chemical runtime* is an essential step towards the model's adoption in the context of autonomic computing. In that sense, we address throughout the thesis the requirements and issues mentioned above both theoretically and practically to deliver a generic, decentralised runtime able to execute chemical programs in large-scale, heterogeneous environments. More concretely, the contribution of the thesis is fourfold: *(i)* the feasibility of such a runtime is explored; *(ii)* an adaptive and efficient protocol for the atomic capture of multiple molecules is proposed; *(iii)* a complete, decentralised chemical platform is presented; and *(iv)* two fully-functional prototypes have been implemented and tested on a real-world large-scale platform, establishing the viability and benefits of the distributed chemical computing approach.

Feasibility Study

We argue that the few isolated attempts at distributing the execution of chemical programs had a limited impact, for they focused on particular types of programs and targeted specific platforms. We therefore conducted a feasibility study of a large-scale runtime for the chemical model by exploring different ways of constructing it. Firstly, an approach to extend the existing centralised runtime, by leveraging the *distributed shared memory* model, is considered. However, after a detailed examination of the limitations inherent to shared-memory models, this approach is abandoned. A second framework is built relying on a peer-to-peer communication layer, on top of which an execution tree structures the

nodes in hierarchy. Each participant executes reactions only with molecules local to it, in this way resolving the problems of molecule discovery and capture. The proposed algorithms are formally shown to have an optimal distribution of termination detection : they do not induce a computational overhead when compared to a centralised version in terms of the number of tests performed.

Atomic Capture of Molecules

On the road towards a decentralised runtime, the most significant barrier to be lifted is ensuring mutual exclusion, at large scale, during the capture of molecules needed by a node in a reaction. Our contribution here is an adaptive and efficient distributed protocol for the *atomic* capture of multiple molecules combining two sub-protocols inspired by previous works on distributed resource allocation, and adapted to the distributed runtime of chemical programs. The first sub-protocol, referred to as the *optimistic* one, assumes that the number of molecules satisfying some reaction's pattern and condition is high, so only few conflicts for molecules will arise, nodes being likely to be able to grab distinct sets of molecules. While this protocol is simple, fast, and has a limited communication overhead, it is not capable of ensuring liveness. The second one, called *pessimistic*, slower, and more costly in terms of communication, ensures liveness in the presence of an arbitrary number of conflicts. Switching from one protocol to the other is achieved in a scalable, distributed fashion and is based on local success histories in grabbing molecules. Furthermore, we analyse chemical programs containing multiple rules and the possible input/output dependencies they might have and propose a rule-changing mechanism instructing nodes as to which rule to execute. A proof of the protocol's correctness is given, while a set of simulation results establishes its efficiency in terms of execution time and network overhead.

Decentralised Chemical Runtime

Built on the lessons learned while conducting the feasibility study and based on the aforementioned protocol for the atomic capture of molecules, a fully-decentralised platform for the execution of chemical programs at large scale is presented. It uses a peer-to-peer network overlay scheme, namely the distributed hash table, to organise the nodes in a ring-like structure and spread the molecules uniformly. On top of this layer, a second is positioned allowing participants to search for candidate molecules in an efficient and scalable manner. The capture protocol is used to grab the reactants atomically. A program's termination is detected in a decentralised way thanks to a molecule classification mechanism separating molecules which have a *reaction potential* from those which do not. We give special consideration to higher-order programs and their execution and incorporate special algorithms to keep the consistency when such programs are dynamically altered.

Proofs of Concept

To complete our theoretical work, two fully-functional prototypes were developed in Java. The first is a prototype implementing the hierarchical, tree-based execution runtime devised for the feasibility study. It improves on the conceptual model by including multiple execution threads, in this way speeding up its performance. The second prototype developed is the practical realisation of the decentralised chemical runtime, enhanced with some optimisations that take the locality of sought objects into account in order to reduce the network traffic generated during a program's execution. Both prototypes where run on a real-world large-scale platform, making them, to the best of our knowledge, the first actual executions of chemical programs in large-scale environments. The experiments conducted comprised up to 1000 nodes. They validate the discussed theoretical frameworks and, thus, present a concrete, practical step towards the adoption of the chemical programming model in specifying autonomic systems deployed in large-scale environments.

Organisation of the Thesis

The thesis is divided in two parts. In the first part, the preliminaries are described and surveyed. More precisely, Chapter 1 delivers the essential ideas and notions the reader should be familiar with in order to appreciate the remainder of the thesis. We describe the chemical programming model focussing on its higher-order variant — HOCL. We briefly overview its syntax and discuss the features related to its execution. As distributed hash tables are on the bottom layer of our solution, we also survey the concepts behind them and describe the different techniques used for object search built on top of them. The last section of the chapter concentrates on works centred around mutual exclusion, particularly its k-out of-M variant, since grabbing multiple molecules bears some similarities to it. Chapter 2 discusses the previous attempts at distributing the execution of chemical programs. First, the uniprocessor runtime is detailed, providing the basic execution algorithm. Then, different existing execution distribution techniques are explained, both those based on message-passing as well as those exploiting the shared-memory model.

Part II of the thesis represents this thesis' contributions. The conducted feasibility study is presented in Chapter 3. We firstly detail the architecture of a runtime based on the distributed shared memory model. After discussing its limitations and drawbacks, both conceptual and practical, we explore a second approach constructed on message-passing techniques. Concretely, we use distributed hash tables for organising the participating nodes in a hierarchical structure and distributing the data and execution. As a part of this platform, we present a distributed algorithm for detecting the execution's termination and formally prove it to be optimal. The developed prototype is then described, laying out its inner workings. The results of the experiments conducted in a large-scale environment are reported. Chapter 4 details the devised adaptive protocol for the atomic capture of multiple molecules. There is an in-depth explanation of the protocol, comprised of two sub-protocols, together with the presentation of its adaptive component — the switch between the sub-protocols. We also discuss a rule-changing mechanism, *i.e.* the behaviour

of the protocol when a program with multiple rules is executed. These concepts are proven formally to be correct, which is further corroborated with simulation results. Finally, the description of the decentralised execution runtime is presented in Chapter 5. We propose an architecture based on two layers — one which stores molecules, the other meta-molecules. Not only does the double-layer structure enable an efficient and intelligent search for molecules, but it also allows the nodes to detect a program's termination in a completely decentralised manner. The details of the prototype are given, after which the experimental campaign conducted on the Grid'5000 test-bed validates the platform's viability and scalability.

Part I

Preliminaries

Chapter 1

Background

Contents

1.1 Chemical Programming Model **20**
1.1.1 GAMMA 20
1.1.2 Higher-order Chemical Language 22
1.1.3 Inertia Detection 27
1.2 Distributed Hash Tables **28**
1.2.1 Overview 29
1.2.2 Some DHTs 30
1.2.3 Range Queries 36
1.3 Mutual Exclusion **39**
1.3.1 Single-resource Mutual Exclusion 40
1.3.2 Multiple-resource Mutual Exclusion 41
1.3.3 *k*-out of-*M*-Mutual Exclusion 43

This chapter introduces the concepts, algorithms and technologies used throughout this thesis. They represent the building blocks of our solution. Firstly, in Section 1.1, we lay out the foundations of the chemical programming model and its variants for which we have built a distributed execution environment. Here we also highlight the significance of the problem of inertia detection. Next, Section 1.2 explains the idea behind distributed hash tables (DHTs) — a technology we rely on for inter-node communication. Finally, Section 1.3 introduces the problem of mutual exclusion of processes in a distributed systems and gives an overview of the current state of the art in the field, with emphasis on the k-out of-M-mutual exclusion problem.

1.1 Chemical Programming Model

With the advent of massively-parallel machines [119], a paradigm able to efficiently exploit the benefits of such architectures was needed. The chemical programming model aimed at filling this role. By being implicitly parallel, its paradigm allows the user to focus on solving their problem without having to deal with issues inherently related to the parallelisation of the devised solution.

In the general description of the chemical metaphor, a *chemical program* can be thought of as a chemical solution. *Molecules*, representing data, float in it according to Brownian motion and collide and react according to some predefined *reaction rules*, constituting the program. The reactions happen in an implicitly parallel, non-deterministic and autonomous manner, in this way removing artificial sequentiality, arbitrary scheduling and structuring from the program. During a reaction, molecules are consumed and new ones are produced (creating new data). This process continues until no more reactions are possible; the chemical solution has at that point reached a stable state known as *inertia*.

1.1.1 GAMMA

The first model to use the chemical metaphor was GAMMA [11]. Following this pioneering programming model, a program is made up of one or more reactions which consume some molecules and produce new ones. The solely data structure available to the programmer is the *multiset* — an unordered set of molecules where each molecule may appear more than once. Molecules can be of any predefined or user-defined type, such as boolean, string, integer, real, *etc*...The main characteristic of the model is the Γ operator, defined recursively as:

$$
\begin{array}{l}
\Gamma(R,A)\,(M) = \\
\quad \textbf{if } \exists x_1,\ldots,x_n \in M \textbf{ such that } R(x_1,\ldots,x_n) \textbf{ then} \\
\quad\quad \Gamma(R,A)\,[(M - \{x_1,\ldots,x_n\}) \cup A(x_1,\ldots,x_n)] \\
\quad \textbf{else} \\
\quad\quad M
\end{array}
$$

M is the multiset on which to operate. The operator R is referred to as the *reaction condition*; it is a boolean function evaluating whether a set of molecules can react or not. The A function describes the result of a reaction. In case the condition holds, the *reactants* are *consumed* — removed from the multiset — and replaced by the outcome of the A function. Reactions, *i.e.* the applications of the Γ operator, are performed until inertia is reached.

It is possible that a reaction condition holds for multiple, disjoint sets of molecules. In that case, their reactions can be carried out simultaneously in an asynchronous manner. Doing so brings the features of *parallelism* (reactions can be carried out simultaneously) and *non-determinism* (there is no predetermined order of execution, since if multiple reactions are possible at the same time, they may or may not be executed as such) to the model. Note that a molecule can be consumed in at most one reaction. If two sets of

molecules satisfying the reaction condition contain the same molecule, then only one of the two sets is actually allowed to enter a reaction.

To illustrate the features of the model and the flow of execution, we are giving here an example proposed in [11]. The program computes the sieve of Eratosthenes, the ancient algorithm for finding prime numbers up to a limit. It can be represented in GAMMA as:

$$\begin{aligned} & sieve(n) = \Gamma(R, A)\ (2, \ldots, n) \text{ where} \\ & \quad R(x_1, x_2) = multiple(x_1, x_2) \\ & \quad A(x_1, x_2) = \{x_2\} \end{aligned}$$

where $multiple(x_1, x_2)$ is true if and only if x_1 is a multiple of x_2. When a pair of molecules is selected from the multiset, it is tested against the reaction condition (R). If one integer is the multiple of the other, a reaction is initiated; the two molecules are removed from the multiset and a new one appears — the one holding the lower value of the two. Hence, while in the original algorithm multiples of a number are *marked* and are not used again in the calculation [95], in the GAMMA version they are removed entirely from the multiset. In this way, once inertia has been reached, the multiset is going to contain only prime numbers up to n.

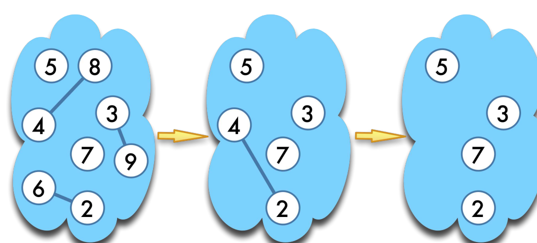


Figure 1.1: Execution of $sieve(9)$.

Figure 1.1 illustrates one of the possible paths the execution of $sieve(9)$ might take. We define an execution step to be a computation cycle in which one or more reactions are carried out. In the first step, three reactions are executed in parallel: 4 and 8 producing 4, 3 and 9 producing 3, and 2 and 6 producing 2. Note that other reactions might have been initiated instead of the ones presented: 2 could have reacted with 4 instead of 6, while 3 could have been coupled with 6 instead of 9. Furthermore, these reactions do not have to happen all at the same time — some might be executed at a later step. Due to the model's non-deterministic nature, the order of the succession of reactions is not defined in advance; the scheduling is left entirely to the runtime executing the chemical program. Finally, the autonomy of reactions provided by GAMMA allows the underlying execution environment to choose the exact location of a reaction at will. That is to say, the three concurrent reactions happening in the first step might all be executed on the same processor (or machine), or they might each be carried out on a different one. The paradigm is not infringed as long as the entity executing a reaction has got the reactants which are not being used in another reaction on another entity at the same time. During the second step the only possible reaction is initiated — 2 and 4 are consumed and 2 is produced — leading the multiset into a stable state; inertia has been reached. The only molecules left in the multiset are those representing prime numbers between 2 and 9, as required.

1.1.2 Higher-order Chemical Language

With its implicitly-parallel paradigm, non-determinism and autonomy of reactions, GAMMA captured the essence of modelling algorithms and programs to be executed on (massively-)parallel machines. In the meantime, however, a new branch of computer science has developed — distributed systems —, bringing with it new problems and challenges. While in the beginning scientists were concerned with classical problems like synchronisation, one of today's challenges is dynamic adaptation: in distributed systems entities appear and disappear constantly and different actors may behave differently during their interactions.

The Higher-Order Chemical Language (HOCL) [99] has been created as a response to these new challenges. Being a successor of GAMMA, HOCL retained its good properties, while raising chemical programming to the higher order. Based on the γ-calculus [12], a higher-order, non-deterministic calculus model, HOCL introduces the notion of *rule molecules*: reaction rules are not treated as special computational entities as they are in GAMMA. Instead, they are regular molecules: they can be consumed and produced in reactions just as regular data molecules can be. The introduction of higher order bears two important consequences:

1. **On-the-fly Program Modification.** Because rules may now be removed or introduced in a chemical solution at run time, the behaviour of the program being executed is able to change over time. In this way, an HOCL program can be dynamically adapted to virtually every aspect regarding its execution: to the environment, to the entities participating in the execution, as well as actors interacting with the program. Furthermore, the program is able to modify itself throughout its execution and thus follow the changes happening in any of the listed participants.

2. **Sequentiality.** Being regular molecules, rules are now placed *inside* solutions, entailing their local reach: a rule can be applied only onto molecules residing in the same (sub-)solution as the rule's molecule. Moreover, as (sub-)solutions are themselves molecules, they can be used in reactions only after their contents is inert. Thus, by putting different rules into different sub-solutions one can achieve the effect of sequentiality.

1.1.2.1 Syntax

The full syntax of HOCL is presented in Table 1.1. A brief overview of its specificities follows.

Reaction Rules. A reaction rule in HOCL is made up of three parts: the pattern of molecules to consume (denoted by P), the product of the reaction (denoted by M) and the reaction condition (denoted by C). The full syntax is:

replace P **by** M **if** C

		Solutions	
S	$:=$	$\langle M \rangle$	*; solution*
	$\mid$	$\langle\rangle$	*; empty solution*
		Molecules	
M	$:=$	x	*; variable*
	$\mid$	M_1, M_2	*; composition of molecules*
	$\mid$	A	*; atom*
		Atoms	
A	$:=$	x	*; variable*
	$\mid$	$[name =]$**replace-one** P **by** M **if** V	*; one-shot rule*
	$\mid$	$[name =]$**replace** P **by** M **if** V	*; n-shot rule*
	$\mid$	S	*; solution*
	$\mid$	V	*; basic value*
	$\mid$	$A_1{:}A_2$	*; tuples*
		Basic values	
V	$:=$	$x \mid 0 \mid 1 \mid \ldots \mid V_1{+}V_2 \mid -V_1 \mid \ldots$	*; standard types*
	$\mid$	**true** $\mid$ **false** $\mid V_1 \wedge V_2 \mid \ldots$	
	$\mid$	$V_1 = V_2 \mid V_1 \leq V_2 \mid \ldots$	
	$\mid$	"string" $\mid V_1$ @ $V_2 \mid \ldots$	
		Patterns	
P	$:=$	$x :: T$	*; molecule with type T*
	$\mid$	ω	*; any molecule*
	$\mid$	$name = x$	*; naming a reaction*
	$\mid$	$\langle P \rangle$	*; inert solution*
	$\mid$	$(P_1 : P_2)$	*; pair*
	$\mid$	P_1, P_2	*; composition of molecules*
		Types	
T	$:=$	B	*; basic type*
	$\mid$	$T_1 \times T_2$	*; product type*
	$\mid$	$\star$	*; universal type*
		Basic types	
B	$:=$	Int $\mid$ Boolean $\mid$ String	

Table 1.1: HOCL syntax (taken from [99]).

A reaction in HOCL involves a set of molecules N, which match the pattern P and satisfy the reaction condition C. These molecules are then consumed, and the set of molecules M is produced. These reaction rules are called *n-shot* rules because their molecules do not disappear from the solution after the reaction. Since in many cases one needs to execute a certain portion of the code only once, HOCL also provides so-called *one-shot* reaction rules, which vanish once they are executed:

replace-one P **by** M **if** C

Reaction rules can also be named, and in this way matched as part of the pattern P of (another) rule. This brings about the possibility of *higher-order reaction rules* — rules which consume or produce rule molecules. Named rules can be *n-shot* as well as *one-shot*. As an illustration, here is the example given in Section 1.1.1, but written in HOCL:

let *sieve* =
  replace x_1, x_2
  by x_2
  if $(x_1 \, mod \, x_2 == 0)$ **in**
    $\langle\ 2, 3, 4, 5, 6, 7, 8, 9, sieve\ \rangle$

Types. As shown in Table 1.1, expressions in HOCL are statically typed using standard types. The language has been designed with flexibility in mind, and thus offers the possibility to put molecules of different types in a solution (reaction rules, integers, *etc.*). Integer, boolean and string constants, along with their associated operations, are supported. Compound molecules — combinations of molecules of different types —- are using the universal type $\star$. More generally, any type is a subtype of $\star$ $(\forall T, T \preceq \star)$.

In complex programs, where there are molecules of multiple types and multiple rules consuming molecules of different types, it is useful to explicitly use types in the pattern P of a reaction rule, such as:

replace $x :: String, y :: Int$ **by** x

This reaction rule specifies that a string and an integer will be consumed, producing a new molecule with the contents of the original string. Thus, if there is no such pair of molecules in the solution, the reaction rule is not going to be triggered. When the type of a molecule is not explicitly stated, it can be inferred from the reaction condition C and the reaction's result set M. An example is the following rule, where the type of x is deduced to be float from the rule's result set:

replace x **by** $\sqrt{x}$ **if** $x \geq 1$

Tuples. A tuple, denoted by $A_1 : A_2 : \ldots : A_n$, represents a compound molecule made up of n atoms. Hence, any valid molecule may become part of a tuple. Tuples are particularly useful when used in conjunction with solutions, as in this way sub-solutions may be isolated from one another.

Matching Any Molecule. Sometimes it is useful to match *zero or more elements of any type*, for example in cases where the goal of a reaction is the removal of a specific molecule or its extraction from an inert sub-solution. This can be achieved with the ω-notation. When specified in a reaction pattern, it matches every molecule in the solution not explicitly specified in the pattern itself. Consider the following reaction rule:

$$eraser = \textbf{replace}\ x :: Int, \omega\ \textbf{by}\ \omega$$

This simple rule effectively removes an integer from the solution: x matches the integer in question, while ω matches all of the other molecules present in the solution, if any are left. Thus, *eraser* can be triggered as long as there are integers in the solution. Note that omitting the type of x might entail a very different execution. Because, in this particular case, the type cannot be inferred from the condition nor the result set, any molecule can be matched as x, including the rule molecule of *eraser*, meaning the reaction rule might remove its own molecule, in this way breaking the (intended) program's logic.

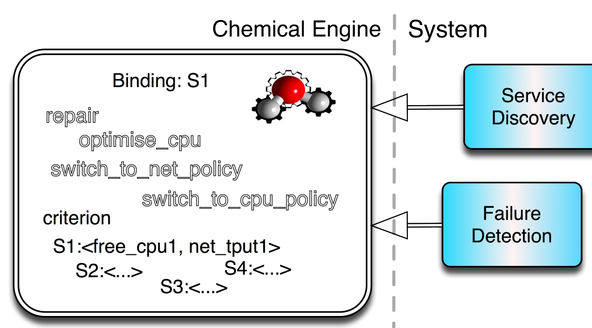


Figure 1.2: An HOCL-based autonomic service.

1.1.2.2 Illustrative Example

We now present a simple, straightforward example showing how HOCL can be used in a dynamic setting which is in need of self-adaptive capabilities. Recall the motivational example given in the previous chapter — the two-tier, autonomic workflow management system. Here we will concentrate on the lower, service management tier.

Let us consider a simple task (or *service*) available on-line and continuously requested by external clients. There exist several concrete implementations (or *services*) on the platform, which can perform this task. In order to be able to handle clients and different services, this component must have the following capabilities:

- *service selection* based on a predefined policy (or criterion), or a set thereof: since there are multiple implementations, the service judged as the fittest (following the predefined policy) has to be active and available to clients;
- *dynamic policy alteration*: due to the dynamic nature of the platform, criteria on which implementations are chosen may change in time (connection issues, spikes in CPU usage, *etc.*), and it is thus essential that the system responds to these changes;
- *failure recovery*: the currently running implementation may fail at any time, in which case the system must be able to switch to another implementation.

Figure 1.2 illustrates the HOCL-based implementation of this self-adaptive service. The multiset (on the left-hand side of the figure) interfaces with two system components, the descriptions of which follow.

- *Service Discovery Unit*. It discovers available service implementations and injects them in the chemical solution as $S_i :< free_cpu_i, net_tput_i >$ molecules. Each such injected molecule represents a service able to perform the task asked by clients. Apart from its identifier (S_i), such a molecule contains a sub-solution indicating some current performance indicators. Here, this sub-solution contains the amount of free CPU ($free_cpu_i$) and the current network load (net_tput_i) of the resource the implementation is running on, but the list of indicators can be arbitrarily extended. These information molecules on services are periodically refreshed.

- *Failure Detector*. This unit detects failures of the currently active service (denoted by the $Binding : S_i$ molecule) and introduces a *"failure_detected"* molecule upon detection.

The rules that drive the execution follow. First, the *repair* rule needs the presence of the specific molecule indicating a failure. Once triggered, it binds the task to another service, removing the previously-binded one:

let *repair* =
  replace $Binding : S_i, S_i :< \omega_i >, S_j :< \omega_j >$, *"failure_detected"*
  by $Binding : S_j, S_j :< \omega_j >$

Let us now review an optimising rule, named *optimise_cpu* :

let *optimise_cpu* =
  replace $Binding : S_i, S_i :< free_cpu_i, net_tput_i >, S_j :< free_cpu_j, net_tput_j >$
  by $Binding : S_j, S_i :< free_cpu_i, net_tput_i >, S_j :< free_cpu_j, net_tput_j >$
  if $(free_cpu_j > free_cpu_i)$

A reaction following the *optimise_cpu* rule is triggered when a service molecule S_j with a better CPU availability is found in the multiset. This rule corresponds to the decision taken by the system to select services based on their CPU availability. Similarly, an *optimise_net* rule may consider a service's network capabilities.

Having different policies brings more flexibility to the adaptation but creates the need for dynamic switching from one to the other based on a criterion, in this case meaning *optimise_cpu* might have to be put aside in favour of *optimise_net*. This is achieved through the higher order, using the following rule, which replaces an optimising policy by another one as soon as it detects the criterion has been altered:

let *switch_to_net_policy* =
  replace *optimise_cpu*, *criterion*
  by *optimise_net*, *criterion*
  if (*criterion* = "Net")

Note that, as illustrated on Figure 1.2, other rules, for instance *switch_to_cpu_policy*, can be similarly constructed and introduced in the solution concurrently. They can coexist smoothly in the solution, as the criterion can take only one value at a time, preventing concurrent reactions of contradictory switching rules. Finally, note that for the sake of simplicity, the example deals with only one service. However, it can be easily extended so as to deal with many services distributed over the nodes of a large-scale platform, each area of the platform having its own criteria and policies changing concurrently.

1.1.3 Inertia Detection

Even though, formally, inertia is simply one of the possible states of chemical programs, in practice it is regarded as a program-termination phase: the execution of a program finishes once inertia has been detected by the underlying runtime environment.

While the notion of detecting inertia is intuitively straightforward to understand, the amount of computation that has to be done in order to achieve varies from program to program as it depends on the rules used. Hankin *et al.* [53] identify five types of rules: reducers, selectors, transmuters, optimisers and expanders. Although each type has its own characteristics, when considering the complexity of inertia detection, they can be grouped in two categories: rules which produce less molecules than they consume (reducers and selectors) and those which produce a number of molecules equal to or greater than the number of molecules they consume (transmuters, optimisers and expanders). We now present a short analysis of inertia detection for reducer and optimiser rules, each a representative of one of the two groups.

1.1.3.1 Reducer Rules

Inertia for reducers is straightforward to detect. Consider the example of a program which finds the maximum integer in a multiset:

let *getmax* =
    replace x, y
    by x
    if $(x \geq y)$
in
    $\langle$ *getmax*, 8, 10, 5, 7, 6 $\rangle$

After every reaction the number of molecules in the solution decreases. Reactions are performed until there is only one integer molecule left, at which point inertia can be detected. Inertia is reached because there are simply not enough molecules present in the solution to perform any more reactions. Hence, detecting inertia in this case is trivial for the underlying execution environment, in the sense that the end of the execution has been reached because there are no more steps which can be performed.

1.1.3.2 Optimiser Rules

When the number of molecules stays the same after each reaction, however, inertia detection is not an easy task. Consider a program sorting an array of elements represented by tuples of the form $i : x$, where i is the element's index and x is its value:

let *sort* =
    replace $i : x, j : y$
    by $i : y, j : x$
    if $(i < j \;\&\&\; x \geq y)$
in
    $\langle\ sort,\ 1:5,\ 2:10,\ 3:15,\ 4:20\ \rangle$

If the two elements are not sorted in ascending order, the *sort* rule consumes them and produces two new molecules with the same indices, but swapped values, keeping in this way the number of molecules in the solution constant, but *optimising* the sorting process.

The solution in the example is already inert; no element is out of place. Nevertheless, to *detect* inertia, the underlying execution environment has to check every possible combination of elements against the reaction condition.

Throughout this thesis we assume, for the sake of simplicity and without loss of generality, that the rules' reaction conditions are commutative functions, *i.e.* that they maintain the same output regardless of the arguments' order ($C(x,y) = C(y,x)$). There is no loss of generality since even if, for a given rule R_1, its condition is not a commutative function, another rule, R_2, can be always constructed such that $C_1(x,y) = C_2(y,x)$ holds. Under these circumstances, the number of combinations to be checked is:

$$N = \binom{m}{r} = \frac{m!}{r!(m-r)!} \tag{1.1}$$

where m denotes the number of molecules in the solution and r is the number of the rule's arguments. Given the number of combinations that has to be checked and given the fact that all of the combinations have to be tried out in order for the runtime to decide inertia has been reached, we are faced with an NP-complete problem equivalent to the well-known *Boolean satisfiability problem* [47]. In spite of the detection's complexity, the execution of chemical programs appears feasible in practice since in most cases the number of a rule's arguments, r, is relatively small, which makes inertia detectable in a reasonable amount of time (as the complexity of N lies in $O(m^r)$).

1.2 Distributed Hash Tables

Due to the vast number of entities participating in its functioning, imposing a centralised architecture on a large-scale application means incurring penalties such as degraded performance, network bottlenecks and failures. Peer-to-peer (P2P) computing offers an alternative to the traditional client-server view in the form of distributed systems which do

not have any hierarchical organisation or centralised control. By having symmetry in the entities' roles, where a client may also be a server and *vice versa*, and by building a fully-distributed cooperative network design with peers forming a self-organising system, P2P systems offer different features such as robust wide-area routing architecture, redundant storage, massive scalability and fault tolerance [87, 77].

Going a step further, distributed hash tables (DHTs) [6] are structured P2P overlay networks which offer efficient, scalable, wide-area data retrieval as well as load-balancing capabilities, enabling them to support the rapid development of a wide variety of Internet-scale applications ranging from distributed file and naming services to application-layer multicast.

1.2.1 Overview

A DHT overlay network is a logical network of participating *peers*, or *nodes*, constructed atop an Internet Protocol (IP) network, the topology of which is tightly controlled and where data is placed at a precisely specified location, determined by a mapping in the form of a hash function. This mapping allows queries to be routed efficiently to the node providing the desired content. Thus, they present a scalable solution for exact-match queries, *i.e.* queries where the exact identifier of the requested data object is known.

1.2.1.1 Operating Principle

Nodes in a DHT overlay network are each assigned a node identifier (a *node id*) chosen from a large set of identifiers. Each data object is referenced by a *(key, value)* pair, as in a traditional hash table. The user provides the value, which is hashed using a uniform, consistent hash function, such as SHA1 [39], to obtain the data's key belonging to the same space of identifiers as that of the nodes. This key is then used to store the data object on a unique peer in the network. Every DHT provides solely two functionalities to its users: one for storing a data object — $put(key, value)$ — and one for retrieving it — $get(key)$.

Each peer maintains a small routing table consisting of a number of known nodes, stored as pairs of node identifiers and their IP addresses, allowing it to choose an appropriate peer to communicate with when resolving requests. Which entries will enter the routing table of a particular DHT depends on its *routing algorithm* and *routing geometry* [52]. The routing algorithm represents the set of mechanisms determining a peer's neighbours (the entries of its routing table) and the next hop in routing a request. In this way, the routing algorithm compels nodes to create an underlying graph with a predetermined organisation — the routing geometry. Depending on a particular geometry, nodes may have more or less flexibility in terms of choosing their neighbours and/or paths for messages.

Hence, every DHT protocol provides a different logic for the *lookup* function — a mechanism allowing nodes to retrieve the node responsible for storing a given key. In spite of their differences, the principle of the mechanism is the same for all DHTs: lookup queries

are forwarded across overlay paths in a progressive manner to peers with node ids *closer* to the key in the identifier space, where *closeness* may be defined differently for different DHT protocols. The lookup function is used to satisfy user-issued $get()$ and $put()$ requests.

In order to preserve the overlay's network topology and routing geometry, two more functions are defined in DHTs, namely *join* and *leave*. When entering a network, a new peer, N, contacts a *bootstrap node*, B — a node for which the new peer knows to be already in the overlay — and asks it to route a special `join` message to node N. This message is routed to node R — the node responsible for N's node id, *i.e.* the node which answers lookup requests for it. The `join` message indicates to R there is a new node to be inserted. It splits its own space with the newcomer N by handing it the part of the keys for which N is going to be responsible from that moment on. N then starts building its routing table, first by copying part of R's routing table and then sending lookup messages on different keys. When N decides to leave the network, it informs its closest neighbours using a `leave` message and transfers them each a part of the keys it holds. The neighbours expand their space by accepting the keys and adjust their routing tables accordingly.

1.2.1.2 Desirable Properties

Due to their design, DHTs exhibit several good properties.

- **Decentralisation.** By splitting the key space amongst nodes, none of which has a global view of the system, DHTs present a natural platform for developing *flat*, fully-decentralised applications where each node is able to change its role — server or client — depending on the computational task at hand and/or on the messages it has received from other participants.

- **Dynamic Environment Support.** DHTs are able to preserve their communication pattern because of the join and leave procedures, and thus are able to support highly-dynamic, volatile environments where participants enter and leave the network at unpredictable moments and possibly do so frequently.

- **Scalable Communication.** Due to the DHTs' routing algorithms and their usage of routing tables, the number of logical hops needed for a message to reach its destination grows logarithmically with the number of nodes; in theory, DHT-based systems can guarantee that any data object can be located in at most $O(n)$ hops, where n denotes the number of peers in the system, since in the worst case, every node is going to be contacted.

- **Load Balancing.** Since the key space in a typical DHT is large in size compared to the number of nodes, they are sparse. However, the usage of uniform, random hashing ensures, with high probability, the uniform distribution of keys amongst nodes, thus balancing the network load during data storage and retrieval.

1.2.2 Some DHTs

We now briefly describe some DHT protocols, each pioneering a different routing geometry.

1.2.2.1 CAN

Geometry. The Content-Addressable Network (CAN) [101] divides a virtual d-dimensional torus into n d-dimensional Cartesian coordinate zones, with n denoting the number of nodes in the system. Thus, each peer is assigned a particular zone, *i.e.* a range of d-dimensional Cartesian coordinates. Keys are deterministically mapped onto a point P in the coordinate space using a uniform hash function. A two-dimensional CAN comprising five nodes is shown on Figure 1.3. Note that the CAN space shown on Figure 1.3 is not a square, since coordinates wrap and the virtual space has no borders. In the special case when $d = \log n$, the geometry of CAN becomes a hypercube.

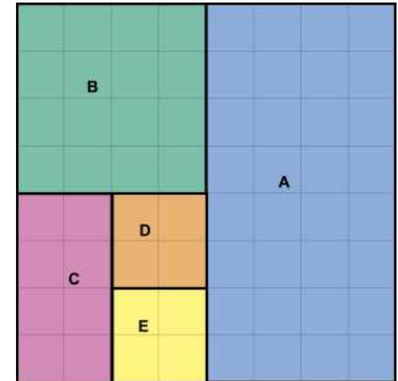


Figure 1.3: Content-Addressable Network (CAN), when $d = 2$.

Routing. The routing table of a node N in CAN comprises only its neighbours — nodes which are responsible for zone adjacent to the one held by N. Messages are routed in a greedy fashion by forwarding them to the neighbour closest to the destination coordinates.

Join/Leave. CAN has an associated DNS domain name, where IP addresses of bootstrap peers are kept. When a new node wants to join the system, the DNS provides it a bootstrap node, which, in return, supplies a list of randomly chosen peers. The new node then randomly chooses a point P and sends a `join` request to one of the previously returned peers. The node responsible for the zone in which P lies splits its zone in two halves (in each dimension), leaving one for itself, and assigning the other half to the newcomer by transferring it the keys belonging to it. The new peer learns of the IP addresses of its neighbour set from the previous peer in point P, and includes it as well.

When a peer leaves the network, an immediate takeover algorithm is triggered, ensuring that one neighbour takes over the zone. It updates its neighbour set and then sends soft-state updates to ensure that all of the neighbours learn about the change. These updates are propagated through the network.

Complexity. For a d-dimensional space partitioned into n equal zones, the average routing path length is $n^{1/d} * \frac{d}{4}$ hops and individual peers maintain a list of $2 * d$ neighbours. The neighbour set does not grow with the increase in the number of peers, while the average path length grows proportionally to $n^{1/d}$. In the case of the hypercube, routing is reduced to $\log n$ hops, and each node has got precisely $\log n$ neighbours, each differing in exactly

one bit. Still, a hypercube-shaped CAN is unlikely in practice since the number of nodes is unknown at network creation time and varies throughout its existence.

1.2.2.2 Chord

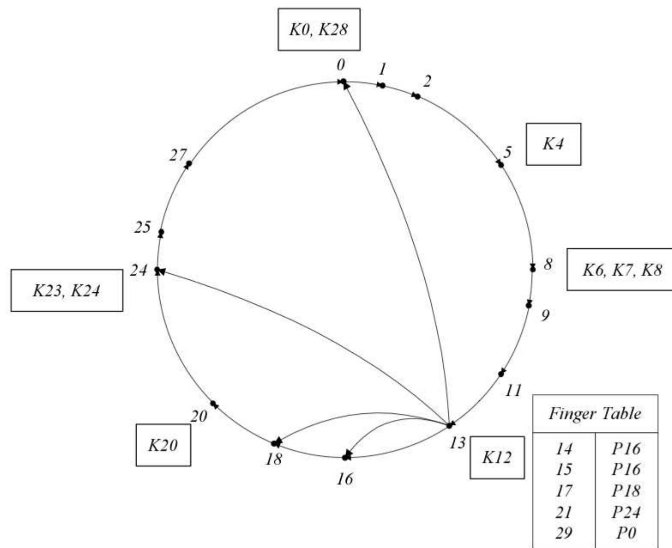


Figure 1.4: Chord ring for $m = 5$.

Geometry. Nodes and keys in Chord [118] are organised into a logical ring in ascending order. Each node keeps a pointer to the node following it, giving rise to the notion of *successor*. The node with the smallest node id is the successor of the node with the highest node id. For a key k, $successor(k)$ denotes the first node in the ring whose node id is higher or equal to k.

Node identifiers and keys are ordered on the identifier circle modulo 2^m, where m is the size, in bits, of identifiers. Chord uses consistent hashing [64] for placing nodes and data along the ring. A peer's identifier is chosen by hashing its IP address and port number, while a key identifier is produced by hashing the data key.

Routing. A node in the Chord ring stores the information about other nodes in the so-called *finger table*. The ith entry in the table of peer N contains the identity of the first peer S that succeeds N by at least 2^{i-1}, *i.e.* $S = successor(N + 2^{i-1})$. A finger table entry includes both the Chord identifier and the IP address, as exemplified in Figure 1.4.

Lookup queries involve the matching of a key and a node id. To route messages a node consults its finger table and forwards the message to the node whose node id is closest to, but less than or equal to, the key. The message is passed around the circle until a pair of peers is encountered whose interval contains the key; the second node is then the message's final destination.

Join/Leave. The join and leave procedures closely resembles the one described in Section 1.2.1.1, with the specificity that a *neighbour* in Chord is a successor. To reduce the time needed to complete the join and leave procedures, each node keeps a pointer to its direct predecessor — the node to which it is a successor. Since the routing heavily depends on nodes' finger tables, the Chord protocol uses a *stabilisation* protocol running periodically in the background to update successor pointers and the finger table entries.

Complexity. A finger table can have up to m entries, but it usually occupies $\log_2 n$ memory locations on each node. Due to the usage of long range contacts, Chord ensures messages are delivered in $O(log_2 n)$ logical hops, with high probability[1] (w.h.p.): at least half of the remaining distance is traversed at each step.

1.2.2.3 Pastry

Geometry. Pastry's [109] key and identifier space is a 128-bit one, and nodes and data are circularly positioned in it, in the range $[0, 2^{128} - 1]$. Node ids are assigned randomly using a uniform hash function, typically SHA1 [39], in this way distributing the nodes uniformly across the space. The same hash function is used to create data keys. Both node ids and keys are considered a sequence of digits in base B, where $B = 2^b$ (b is a configuration parameter and is usually set to $b = 4$).

In spite of the circular identifier space, Pastry's routing geometry is actually hybrid, since a tree-like node organisation is used for routing, where each node constructs its own tree of which it is the root and any node sharing its node id prefix with it is a candidate for inclusion. More precisely, nodes placed on level i of the tree share all but the last i digits with the root's node id.

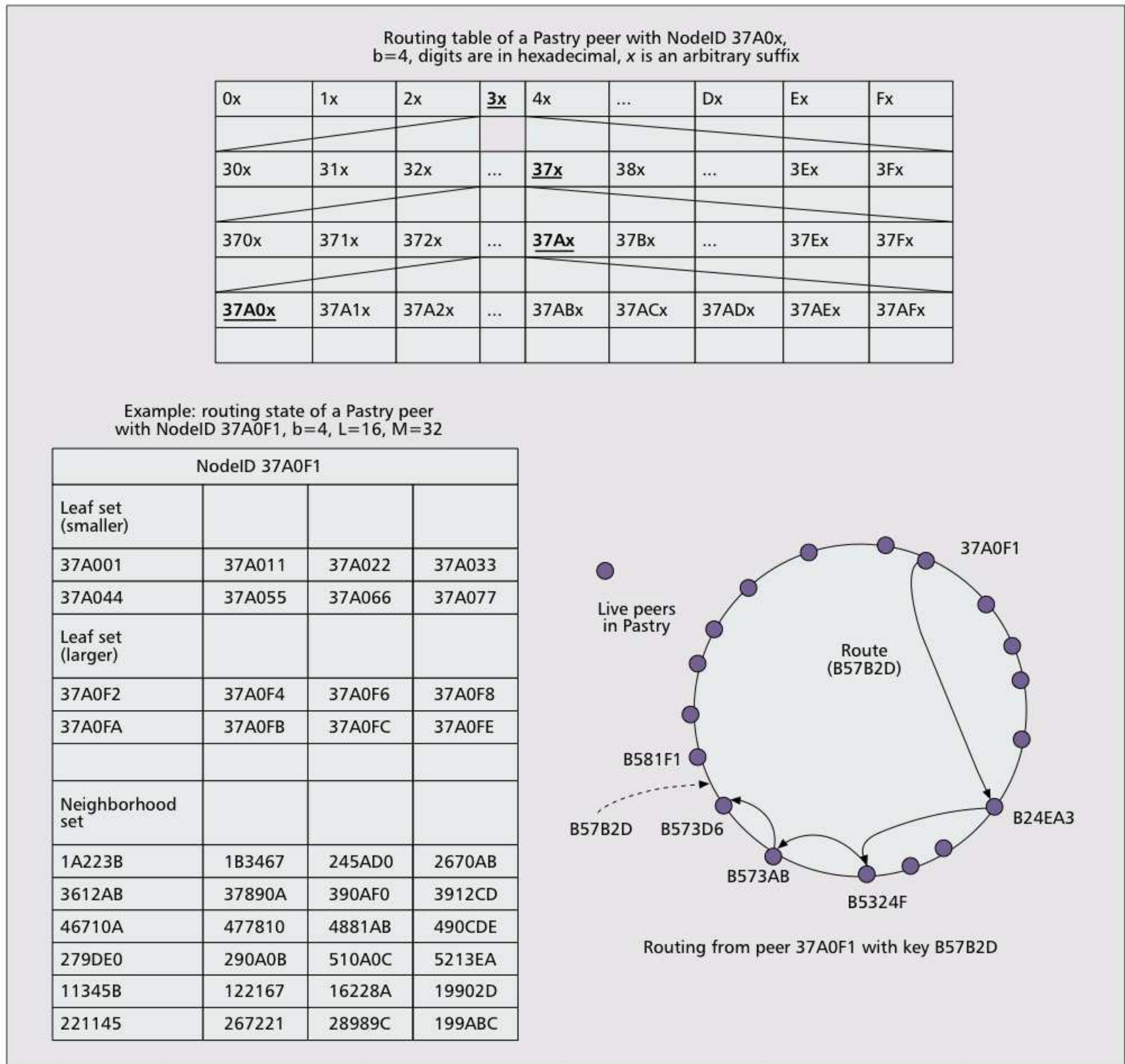
Routing. Pastry is a derivative of the prefix-based routing algorithm proposed by Plaxton, Rajaraman and Richa [97]. Thus, messages are routed so that the node in the next hop shares a bigger common prefix with the message's key than the current node. The final destination is the node the node id of which is numerically the closest to the key. Pastry nodes divide their routing information in three parts: a routing table, a neighbourhood set and a leaf set. An example of the contents of these structures is shown in Figure 1.5.

- *Routing Table.* The routing table basically contains the routing tree used by the node. There are $B - 1$ entries per table row, where each entry is composed of a node id and its IP address. Row i contains a node if its node id shares the current node's identifier's first i digits, but its $(n+1)^{\text{th}}$ digit has one of the $B-1$ other possible values.

- *Neighbourhood Set.* In order to improve the routing performance, each node maintains a neighbourhood set, M, populated by $|M|$ nodes closest in proximity to the current node. The network proximity used in Pastry is based on a scalar proximity metric such as the IP-routing geographic distance. The neighbourhood set denotes a node's physical neighbourhood.

- *Leaf Set.* The leaf set, L, is the set of nodes with identifiers close to the current node's. The set is split in two subsets: $\frac{|L|}{2}$ nodes with smaller identifiers, and $\frac{|L|}{2}$ nodes with greater node ids. A node uses the leaf set to determine exactly which keys belong to it and which belong to its neighbours. The leaf set represents a node's logical neighbourhood.

[1] An event occurs *with high probability* if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which the event occurs with probability at least $1 - O(1/n^{\alpha})$, where n denotes the number of nodes


Routing table of a Pastry peer with NodeID 37A0x, b=4, digits are in hexadecimal, *x* is an arbitrary suffix

0x	1x	2x	3x	4x	...	Dx	Ex	Fx
30x	31x	32x	...	37x	38x	...	3Ex	3Fx
370x	371x	372x	...	37Ax	37Bx	...	37Ex	37Fx
37A0x	37A1x	37A2x	...	37ABx	37ACx	37ADx	37AEx	37AFx

Example: routing state of a Pastry peer with NodeID 37A0F1, b=4, L=16, M=32

NodeID 37A0F1			
Leaf set (smaller)			
37A001	37A011	37A022	37A033
37A044	37A055	37A066	37A077
Leaf set (larger)			
37A0F2	37A0F4	37A0F6	37A0F8
37A0FA	37A0FB	37A0FC	37A0FE
Neighborhood set			
1A223B	1B3467	245AD0	2670AB
3612AB	37890A	390AF0	3912CD
46710A	477810	4881AB	490CDE
279DE0	290A0B	510A0C	5213EA
11345B	122167	16228A	19902D
221145	267221	28989C	199ABC

Figure 1.5: Routing table, neighbourhood set and leaf set of a Pastry peer, with an example of routing (taken from [77]).

When a node receives a message, it first consults its leaf set. If the message's key falls inside the leaf set's range, the final destination node is determined and the message is sent to it; the node itself handles the message if it happens to be the node with the numerically closest identifier to the key. Otherwise, the node refers to its routing table to find the node with the longest common identifier prefix as the message's key. If a node with the same prefix length exists in the neighbourhood set, it is preferred over the one found in the routing table.

Join/Leave. When joining the Pastry ring, node X contacts an *a priori* known node A and asks it to route a `join` message with the key equal to X. The message is routed to node Z which is responsible for the key X. Upon receiving the join request, nodes A, Z and all others encountered on the path send their routing tables to node X. Using this information, it initialises it structures and informs any nodes which need to be aware of its arrival.

Pastry has got a dynamic fault-tolerance mechanism. Nodes periodically check the state of nodes in their leaf set. When a node perceives one of them is not responding, it contacts the live node with the largest index in the subset of the failed node and requests its leaf set. The failed node is replaced by an alive node residing in the set $L' \setminus L$, where L' denotes the received leaf set. In the same vein, the neighbourhood set is periodically checked. If a node is not responding, the contacting node asks other members of the set for their neighbourhood sets. It then finds the node closest to it, as per the proximity metric, which is not already part of the neighbourhood set and replaces the failed node with it.

Complexity. The routing table has an average of $\lceil \log_B n \rceil$ rows with each holding $B-1$ entries. The leaf and neighbourhood sets are of tunable sizes, but their typical sizes are B and $2 * B$, respectively. Since the path length between two nodes is determined by the sub-tree connecting them, Pastry needs at most $\log_B n$ logical hops to route a message, w.h.p. The routing algorithm's efficiency is further enhanced with good locality properties, since nodes will select as neighbours nodes closer to them as per the proximity metric.

1.2.2.4 Other DHTs

Viceroy. By enhancing Chord with carefully selected long range contacts, Viceroy [80] maintains an architecture which is an approximation of the butterfly network [113], making its geometry hybrid. A Viceroy network is comprised of three different types of structures: *(i)* a Chord-like ring; *(ii)* the butterfly, connecting nodes from the Chord-like ring using an emulation of the butterfly network; and *(iii)* level rings, bringing together nodes from the same butterfly level. Due to the simultaneous maintenance and usage during routing of all of these structures, Viceroy exhibits a constant out-degree and logarithmic diameter. Its diameter is better than CAN's and its degree is significantly lower than that of other ring-based DHTs (Chord, Tapestry, Pastry).

D2B. With a geometry based on a de Bruijn graph [33], D2B [43] is able to keep the node degree constant, while at the same time offering lookup operations which complete in $O(\log n)$ logical hops. The keys are uniformly distributed with an upper bound of $O(|\mathcal{K}| \log n/n)$ keys per node, where $\mathcal{K}$ denotes the set of keys. The basic version of D2B uses a de Bruijn graph of dimension 2. However, it is also possible to construct a geometry with an arbitrary dimension d while keeping the interesting properties. The drawback of D2B stems from the fact that identifiers are statically assigned to nodes, limiting its usage in real, dynamic settings.

eCAN. Expressways-CAN (eCAN) [126] is an extension to CAN which optimises path lengths by creating *shortcuts* inside the CAN torus. The main idea is to divide the CAN space into k subspaces and continue doing so for each subspace for as many levels as wanted. For each division, a node creates a link to one node in each adjacent subspace. By keeping such long range contacts, routing path lengths are shortened when a message passes a subspace barrier. This way, the routing process needs $\ln n$ hops to complete.

Tapestry. Tapestry [127] is in its essence very similar to Pastry. It is also inspired by the prefix-based routing from Plaxton *et al.* [97], and has, thus, a complexity and performance comparable to that of Pastry, despite the fact that Tapestry resolves digits in the opposite order. However, nodes in Tapestry do not maintain a leaf set like in Pastry. Consequently, its routing geometry is a pure tree. Another significant difference is the usage of the neighbourhood set. Tapestry automatically puts closer nodes into the routing table, and in doing so it limits the number of distinct routes a message can traverse to its destination.

Kademlia. Like Chord, Kademlia [83] relies on the notion of distance between two identifiers, but instead of the Euclidean metric, it uses XOR. XOR is symmetric and it allows nodes to receive lookup queries from precisely the same distribution of nodes contained in their routing tables. Furthermore, XOR has the property of being unidirectional, which makes sure all of the lookups for the same key converge along the same path. Like Pastry, Kademlia exploits locality by routing messages towards one of the k peers closest to the destination based on latency.

Koorde. Koorde [61] is a DHT that augments the basic Chord ring with a de Bruijn graph. While sharing some similarities with D2B, most notably the network geometry and constant out-degree, Koorde differs from D2B in that it uses Chord's scheme for node id assignment and handling of multiple joins, while D2B forces nodes to adopt a particular identifier in order to construct a de Bruijn graph. Koorde achieves the following optimal results: *(i)* for a $O(1)$ node degree, the network's diameter is $O(\log n)$ w.h.p.; and *(ii)* for a $O(\log n)$ node degree, the diameter is $O(\log n/ \log\log n)$.

1.2.3 Range Queries

Because DHTs are *data-centric* systems, one of their major shortcomings is that they primarily support exact-match, single-key queries. Indeed, the very feature that makes for their good load-balancing properties — the usage of randomised hash functions — works against range queries. Hence, additional mechanisms have to be employed to provide for such a functionality.

Since resolving complex queries, and in particular range queries, is a feature needed in every information-retrieval system, there are many proposals in the P2P community which tackle it in different ways and by using different underlying structures [54, 9, 44, 19, 32, 122]. We here focus on solutions which build upon DHTs and give a short overview of the most significant ones.

MAAN. The Multi-Attribute Addressable Network (MAAN) [21] was one of the first systems enabling range queries in DHTs. Based on Chord, MAAN provides the mechanisms for multi-attribute range queries by essentially replacing SHA1 with a uniform locality-preserving hash function. Objects are placed along the Chord ring so that their order is preserved with regards to a specific attribute. For resolving a range query, a request is sent to the node responsible for the range's lower bound. It adds the objects matching the criteria to a list and passes it on to its immediate successor. This process is repeated until the query reaches the node responsible for the range's upper bound, which sends the result back to the originating node. Hence, MAAN needs $O(n)$ messages in the worst case. When multiple-attribute objects are stored in the DHT, a hash value is calculated for each attribute value and the object is stored at multiple locations. Resolving a multiple-attribute query involves sending m sub-queries iteratively, where m is the number of attributes the range query contains.

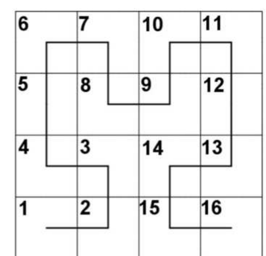


Figure 1.6: 2-d Hilbert curve, second refinement level.

Inverse Space-filling Curve. By using the inverted Hilbert's space-filling curve [7], Andrzejak et Xu [5] map a single-attribute domain onto CAN's generalised d-dimensional Cartesian space (Figure 1.6). They assume objects have a single, continuous attribute whose values fall in the range $[0, 1]$. Since Hilbert's algorithm translates a point in the d-dimensional space into a value in the same range while preserving data locality[2], Andrzejak and Xu simply invert it to place an object on the CAN torus. Resolving a range query involves sending a request to the node responsible for the point in the middle of the range, which then searches its surroundings along the curve to fulfil the query. The authors propose three methods in the paper: *(i)* brute force, which computes the smallest hypercube encompassing all of the zones intersecting the range; *(ii)* controlled flooding, in which messages are sent to nodes that *definitely* intersect the query; and *(iii)* direct controlled flooding, which floods the right and then the

[2]Data locality is preserved in the sense that data points which are close in the original d-dimensional space will remain close after the translation.

left side of the curve in search for a match.

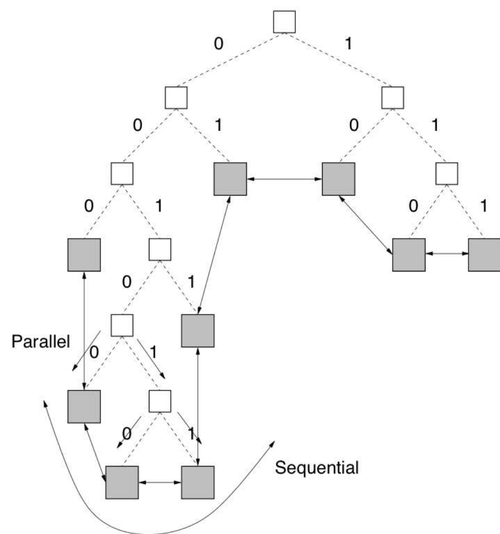


Figure 1.7: Range-query resolution using a PHT.

Prefix Hash Tree. The Prefix Hash Tree (PHT) [100] is a trie-based distributed data structure built relying exclusively on the *lookup* function, and can, thus, run atop any DHT. Each node of the PHT is labelled with a recursively-defined prefix: given a node with label l, its child nodes are labelled $l0$ and $l1$. The PHT is mapped to the DHT by hashing each PHT node's prefix label and locating it on the node responsible for $hash(l)$. A data object is found by performing a DHT lookup on its prefix. Keys are stored only on leaf nodes, which form a double-linked list, used in the sequential traversal for answering range queries — given the bounds of a query, $[l, u]$, the PHT locates $leaf(l)$ and traverses the list until $leaf(u)$. The paper presents another, parallel, algorithm for resolving range queries. Using the DHT, the node whose prefix covers the whole range is located. It then recursively forwards the query to those children which overlap with the specified range. The two algorithms are illustrated in Figure 1.7. To store multi-attribute objects and resolve queries over them, the PHT uses space-filling curves to map a d-dimensional object onto a one-dimensional point and then references it during operation.

Range Search Tree. Like the PHT, the Range Search Tree (RST) [45] can operate on top of any DHT. An RST is a complete and balanced binary tree with k leaf nodes and $\lceil \log k \rceil + 1$ levels. An attribute's domain is split in k sub-domains, one per leaf node, while the sub-domain of a non-leaf node corresponds to the union of its children's sub-domains. For each sub-domain, the RST keeps a *load-balancing matrix* (LBM) — a set of nodes responsible for the sub-domain. RST nodes are mapped onto DHT nodes by hashing the RST node's attribute, its sub-domain and the column and row indices in its LBM. A range query, with the range length R_q, is decomposed into $O(\log R_q)$ disjoint sub-queries, each corresponding to a node in the RST. Each sub-query is then resolved by the matching node in a top-down fashion.

Squid. Another system using Hilbert's space-filling curve is Squid [111]. It was the first range-query system which *natively* supported multi-attribute range queries. Squid supports keyword searches, including wildcards, partial keywords and ranges, and guarantees that the information will be found if it exists in the system. Each data element that is indexed is described using a set of keywords. All the possible keywords form a multi-dimensional keyword space and each data element can be regarded as a point in this space. Hilbert's curve is then used to map this multi-dimensional keyword space to a one-dimensional index space. It is DHT-independent as it does not require any modification

of the underlying DHT. The mapping between Squid indices and DHT nodes is straightforward, because the index space is chosen so as to match the DHT's key space. To process a range query, the keyword-based query is translated into one-dimensional-point segments — clusters (contiguous ranges). While in a simple case, sub-queries matching these clusters might be directly sent to the relevant nodes, the authors of the paper offer an optimised, distributed querying procedure, in this way balancing the query-resolving load. The query in question is refined by following Hilbert's curve's refinement process. At each refinement step, as the query gets divided, its sub-queries are sent to the nodes responsible for the given region of the d-dimensional space. Effectively, a *refinement tree* is created for each query. The authors show that, for a generic query matching $p\%$ of the data, the number of nodes with matching data approaches $p\%$ of all nodes in the system.

1.3 Mutual Exclusion

One of the fundamental research topics in distributed systems has been mutual exclusion of processes competing for shared resources. The access to such a shared resource must be synchronised amongst processes (or nodes) to ensure that only one process is making use of the resource at a given time. Each process must request permission to enter its critical section and must release it after it has completed its execution. Therefore, a mutual exclusion algorithm must meet the following requirements [114]:

- *safety*: at most one process can execute its critical section at a given time;
- *liveness*: if no process is in its critical section, any process requesting to enter its critical section must be allowed to do so in finite time;
- *starvation*: when competing processes concurrently request to enter their respective critical sections, the selection cannot be postponed indefinitely;
- *fairness*: a requesting process cannot be prevented by another one to enter its critical section within a finite delay.

Distributed mutual exclusion algorithms can be roughly divided in two categories [105]:

- **Permission-based.** In the permission-based group the right to enter a critical section is formalised by receiving permission from a set of nodes in the system. A process wishing to enter its critical section asks the others to give it their permission to proceed; and then it waits until these permissions have arrived before entering the critical section. Each process may grant its permission to only one process at a time.
- **Token-based.** In the token-based group the right to enter a critical section is materialised by a special object, a *token*, unique in the system. The singular existence of the token implies the enforcement of mutual exclusion. Processes requesting to enter their critical section are allowed to do so when they possess the token. After a process has finished executing its critical section, it chooses the next token owner and sends it the token.

Regardless of whether a particular mutual exclusion algorithm is permission- or token-based, it can tackle the problem of allocating only one resource to a process at a time, or multiple ones. We here give a short overview of these two kinds of algorithms and then concentrate on k-out of-M-mutual exclusion, a special kind of multiple-resource allocation problem. Note that throughout this section we use the terms "node" and "process" interchangeably.

1.3.1 Single-resource Mutual Exclusion

1.3.1.1 Permission-based Algorithms

The first distributed mutual exclusion algorithm proposed uses *timestamps* — event counters accompanied by the identifier of the process generating it — to define the total order in a set of processes competing for a resource [70]. A process entering a critical section sends request messages to all of the other processes. When a process receives a request message, the request is put into a priority queue and it sends a reply message to the requesting process. The requesting process enters a critical section if it receives reply messages from the other processes and its request is the highest among items in its priority queue. To exit from a critical section, it sends a release message to the other processes and deletes its request from its queue. This algorithm is based on the unanimous consensus method and requires $3(n-1)$ messages per mutual exclusion cycle.

Ricart and Agrawala [106] improved it by removing the sending of messages when the process exists a critical section, thus achieving a total cost of $2(n-1)$ messages per cycle. Carvalho and Roucairol [25], in their turn, further improved the algorithm. The number of messages with their approach varies from 0 to $2(n-1)$. The reduction in messages comes from the simple observation that if node i grants its permission to node j, j does not have to ask i's permission again at a later time, unless i wants to enter the critical section before j does. By analysing the aforementioned algorithms, Sanders introduces the concept of an information structure as their unifying principle and proposes a generalised mutual exclusion algorithm based on it [110].

Thomas noticed that in order to enter its critical section, a process does not necessarily have to obtain the permission of *all* of the others. He proposed the majority consensus algorithm [121], where permission has to be granted by only $(n+1)/2$ processes, thus automatically cutting in half the number of messages. Mutual exclusion is still guaranteed: any two sets of $(n+1)/2$ processes each will have at least one process in common, and since that process can give its permission to at most one process, no two processes can enter their critical sections at the same time. Variations of this algorithm have also been proposed, such as weighted voting [49] and multidimensional voting [30].

As a generalisation of majority and voting, Garcia-Molina and Barbara developed the notion of a *coterie* [46] — a set of *quorums*, sets of nodes, where each pair of quorums has got a non-empty intersection and no quorum is a subset of another. Using quorums decreases the number of messages since a process wishing to enter a critical section sends request messages only to processes in a quorum. Coteries can be constructed by using finite projective planes [79], binary trees [4] or multilevel hierarchies [68]. Neilsen,

Mizuno and Raynal proposed a general way of constructing coteries, and they show how to design complex coteries using simple ones [92].

1.3.1.2 Token-based Algorithms

Ricart and Agrawala were the first to propose a token-based algorithm [107]. Entrance to the critical section is moderated by an incremental sequence counter. If a node wishes to enter its critical section, it sends a token request to every other node and waits until one of them sends it the token. By implementing a request array holding the sequence numbers of the last requests by other nodes, after releasing the token, a node knows to which node to send the token next. This algorithm requires either n messages, when a node requests the token, or 0 messages, when a node currently holding the token wants to re-enter its critical section. A similar approach was presented by Suzuki and Kasami [120], where a queue is used instead of an array.

Their algorithm was the basis for the Mizuno-Neilsen-Rao algorithm [88], which incorporates *quorum agreements* to form groups of nodes inside the network. Another algorithm based on sequence numbers and complete system states is the one proposed by Singhal [115]. It uses heuristics based on states received from other nodes to reduce the number of message exchanges; on average, $(n+1)/2$ messages are sent, with an upper bound of n messages. The Suzuki-Kasami algorithm is the basis of the algorithm devised by Nishio, Li and Manning [94]. In their variant, the token is given to the nearest node requesting it. This allows them to impose time-outs on requests, in this way enabling fault tolerance.

Neilsen and Mizuno [91] impose a static logical structure on the communication network where nodes are arranged in a directed acyclic graph and communicate only with their neighbours. A node does not have to maintain a queue of pending requests. This queue is implicitly maintained by the state of each node in the system. The total number of messages exchanged per critical section entry depends on the topology of the logical structure, but has an upper bound equal to $d+1$, where d is the diameter of the network. Similarly, Raymond proposed an algorithm where the nodes form a spanning tree and communicate only with their neighbours [103]. Since the token can travel only along the tree's edges, the algorithm requires typically $O(\log n)$ messages per cycle. Naimi and Trehel [90] arrange the nodes dynamically in a rooted tree based on their requests: as a request from node i travels along the path from node i to the root node, node i becomes the new parent of each node on the path, except for itself. Thus, node i becomes the new root node of the tree eventually.

1.3.2 Multiple-resource Mutual Exclusion

Sometimes processes can or have to access multiple resources at the same time, for example when there are multiple instances of a resource or when a node needs multiple resources in order to complete its task. Hence, different variants of the multiple-resource mutual exclusion problem exist:

- *Dining philosophers.* Originally addressed by Dijkstra [35], the dining philosophers problem refers to situations where a process competes for a fixed set of resources with an *a priori* known set of processes. The original proposition gives the analogy of five philosophers at a round table, each competing with its two neighbours over forks.

- *Drinking philosophers.* Chandy and Misra generalised the dining philosophers problem in the drinking philosophers problem [28]. Instead of forks, now the philosophers compete for bottles. A philosopher is not limited at drinking only from the bottle shared with his neighbour, but may choose any number of bottles. Moreover, he might need different sets of bottles during different drinking sessions.

- *Group mutual exclusion.* In the group mutual exclusion problem, each process is assigned to a group. Processes may enter their critical sections simultaneously if and only if they belong to the same group. Continuing the philosophers analogy, Joung formulated the problem in terms of congenital talking philosophers [60]. There are n philosophers and m forums, but only one meeting room. A philosopher interested in a forum may enter the meeting room in two cases: if the meeting room is empty or some philosophers interested participating in the same forum are already present in the meeting room.

- *k-mutual exclusion.* First formulated by Fischer *et al.* [42], the k-mutual exclusion problem, or k-mutex, addresses situations where k identical copies of a resource exist simultaneously in the system. With the resources being identical, up to k processes may be given permission to enter their critical sections.

- *k-out of-M-mutual exclusion.* A generalisation of k-mutex was given by Raynal in the form of the k-out of-M problem [104]: on some occasions, a process may wish to access k out of the M existing copies of a resource.

In the remainder of the section we will present works dealing with k-mutual exclusion, while the next section formalises the k-out of-M-mutual exclusion problem and offers an overview of works tackling it.

k-Mutual Exclusion

Raymond was the first to provide a solution to the k-mutex problem [102]. Her solution is permission-based and is based on the algorithm provided by Ricart and Agrawala [106] for the distributed mutual exclusion problem. As in the case of Ricart and Agrawala's algorithm, a node seeking access to its critical section sends a request message to all of the remaining $n-1$ nodes, but it enters the critical section as soon as $n-k$ nodes reply. The algorithm's message complexity varies between $2n-k-1$ and $2(n-1)$.

Srimani and Reddy [117] proposed a token-based solution. They introduce k tokens into the system. A node seeking access to the critical section sends request messages to all of the nodes and enters it only after obtaining a token. The existence of k tokens in

the system enables up to k processes to access their critical sections simultaneously. The worst-case message complexity of the algorithm is $n+k-1$. The algorithm is analogous to that of Suzuki and Kasami [120] for the distributed mutual exclusion problem, in which the system has a unique token. Special provision has been made in the algorithm to take care of any possible starvation problems arising due to the presence of multiple tokens.

Kakugawa *et al.* [62] introduce the concept of a k-coterie on a non-empty set $\mathcal{V}$ as being a collection $\mathcal{C}$ of non-empty, minimal subsets of $\mathcal{V}$, called quorums, and chosen in such a way that it is always possible to find, starting from any quorum of $\mathcal{C}$, a group of k mutually disjoint quorums in $\mathcal{C}$, but never possible to find $k+1$ mutually disjoint quorums in $\mathcal{C}$. In [63] they present a detailed distributed k-mutual exclusion algorithm based on k-coteries, analogous to the one of Maekawa [79] used to solve 1-mutual exclusion. However, unlike Maekawa's algorithm in which quorums are determined statically, a node seeking access to the critical section searches for a quorum dynamically until it succeeds in obtaining permission from each and every node of some quorum of the k-coterie or until the possibility of finding such a quorum is ruled out.

1.3.3 k-out of-M-Mutual Exclusion

Consider a set of M identical resources shared by n processes (or nodes). At any given moment each of the n processes may wish to use k ($1 \leq k \leq M$) resources exclusively. It has to obtain the right to use them at once and thus it remains blocked until it is granted access to all of them. A conflict arises when a process tries to allocate more resources than there are available ones. Hence, the k-out of-M problem lies in ensuring the correct management of these M resources such that two properties are satisfied:

- *safety*: at any given moment, the number of resources allocated to the processes is less than or equal to M;
- *liveness*: all requests have to be satisfied within a finite amount of time.

The problem is a generalisation of both the 1- and the k-mutex problem. Indeed, when limiting the number of resources M to 1, the problem is reduced to the classic 1-mutual exclusion one, while when fixing k to 1, the problem reflects k-mutex.

1.3.3.1 Raynal's Algorithm

Raynal [104] formulated the problem and provided a simple solution. It is a permission-based algorithm based on the one given by Ricart and Agrawala [106]. To satisfy the two properties listed above, the algorithm uses timestamps, which are put on requests, and an array holding the information about the quantity of resources used by each node.

Liveness is ensured by imposing a total order on the requests via timestamps and serving them according to this order. In order to ensure safety, each process is given a consistent view of the number of resources used by all of the other processes. This information is kept in an array denoted *used*. Safety is guaranteed by imposing the following invariant condition on the array:

$$\sum_{1 \le j \ne i \le n} used[j] + k \le M$$

where i denotes the identifier of the process at which the condition is being checked. In other words, consistency is achieved if and only if on every node the sum of the number of resources used by all of the nodes is less than or equal to the total number of resources (M).

The algorithm itself is pretty straightforward. To obtain resources, a node sends timestamped requests to all the others in the system. Because any of them may be using an arbitrary number of resources, the node increases each of their values in the *used* array by M, in this way enforcing the safety condition. As each reply includes the number of free resources perceived by its sender, the node updates its *used* array by subtracting the received number for each node from each array element. If the safety condition holds, *i.e.* if the sum is less than or equal to M, the node proceeds to use the resources. When a node receives a request from another node, it sends back a `free`(M) message if it does not need the resources or if the received request has got a higher priority. Otherwise, it replies with a `free`($M-k$) message, k being the number of resources the node is interested in, indicating to the requester that it can use at most $M-k$ resources. At a later time, once the node has finished using the resources, it will send another message to the requester, `free`(k), notifying it the resources have been freed.

The algorithm assumes a fully-connected network, even though it can be adopted to other topologies as well. Furthermore, communication channels are assumed to be reliable and the order of messages is preserved. The algorithm described uses between $2(n-1)$ and $3(n-1)$ messages for each entrance to the critical section. However, in the same paper, Raynal proposes an optimisation which reduces the lower bound on the number of messages to 0. He observes that if an element of *used* is non-zero, there is no need to query that node, so no request is sent to it.

Additionally, Raynal introduces the generalised AND-allocation problem. He considers several types of resources, $R_1, R_2, \ldots, R_p$, and each resource type is represented by several identical resources: there are M_α instances of resource type R_α. A node asks for all of the resources it needs at once, regardless of their type, and it remains blocked until they are allocated to it. In order to solve the problem, the k-out of-M algorithm has to be slightly adapted, only to accommodate simultaneous requests for different resource types. As nodes have distinct identities, giving a unique timestamp to each demand allows resolving all of the conflicts in the favour of the same node whatever the resources concerned.

1.3.3.2 Quorum-based Algorithms

Even though simplistic, Raynal's algorithm's efficiency can be improved since a node has got to obtain the permission of every other node in the system. The algorithms described below in most part adopt Maekawa's usage of quorums [79] and apply it to the k-out of-M problem in order to minimise the number of permissions a node has to obtain before entering its critical section, in this way reducing the number of messages. In each of them,

a node asks the permission of a quorum and executes its critical section only if it has received positive replies from all of the nodes in that quorum. Upon releasing the resources, the node notifies the members of the quorum. The algorithms differ in the structures used as well as in the way quorums are constructed and in their size, directly influencing the complexity in terms of number of messages.

Arbiter Sets. Based on Maekawa's concept of an arbiter, Baldoni [10] introduces *arbiter sets* ($\mathcal{A}$), one per process, which are derived from quorums: a process, *p*, *arbiters*, *i.e.* grants permissions to, requests from processes in its arbiter set, $\mathcal{A}_p$ — the set of processes which have sent *p* a request. An arbiter set is split in two sets on each node: $\mathcal{S}_p$, the set of processes which are using some resources, and $\mathcal{Q}_p$, the set of processes waiting on resources. When a request for *k* resources arrives from process *r*, *p* puts *r* in $\mathcal{S}_p$ and grants it its permission if the processes in $\mathcal{S}_p$ are using less than or an equal number of resources to $M - k$. Otherwise, *r* is put in $\mathcal{Q}_p$. *r* can proceed in using the *k* resources only once it has received the permission of all of the processes in its quorum, *i.e.* those to which it previously sent the requests. Baldoni uses the combinatorial approach to calculate that the minimum cardinality of a symmetric arbiter set lies in $O(n^{M/(M+1)})$. Due to symmetry, that is also the size of a quorum, and hence, the number of messages lies also in $O(n^{M/(M+1)})$, which represents only a slight improvement.

***M*-Arbiters.** Following Baldoni's work, Manabe *et al.* [81] formally define an *M-arbiter* as a set of quorums where: *(i)* no quorum is a subset of another; and *(ii)* the intersection of any $M + 1$ quorums is non-empty. The mutual exclusion algorithm given in the paper is identical to Baldoni's, but the authors provide examples of *M*-arbiters, such as singleton, uniform, $(k + 1)$-cube and symmetric *M*-arbiters, which can be used to produce different quorum sets. However, quorums in *M*-arbiters are much larger than their equivalents used for 1-mutual exclusion, entailing that the algorithm consumes more messages than needed.

(k, M)-Arbiters. Manabe and Tajima [82] pointed out that the *M*-arbiter algorithm uses the same quorum regardless of the request. They observed that when a process requests a small number of resources (close to 1), it has to be aware of the existence of a large number of other requests in order to detect a conflict, but when it needs a high number of resources (close to M), it can detect its request is blocked by knowing only a small number of other requests. Hence, they introduce a (k, M)-arbiter, denoted $\mathcal{Q}_{k,M}$, which comprises *M* quorum sets — one for each *k*, $1 \leq k \leq M$. The authors show that the quorums in each quorum set of a (k, M)-arbiter are not larger than those in a corresponding *M*-arbiter. At the same time, they cannot be smaller than the quorums used in 1-mutual exclusion. The algorithm proposed in the paper differs from Baldoni's in two regards. Firstly, before sending out requests, a process chooses a quorum based on the number of resources it is about to demand. Secondly, apart from keeping a served and a waiting queue, each process maintains a third set for requests which are being cancelled at the moment. A process may be cancelled if, after it has been given permission to use the resources, a

request with a higher priority arrives. Keeping this additional set allows them to further reduce the number of messages.

Chapter 2

Physical Parallelism of the Chemical Model

Contents

2.1 Single-processor Execution . . . 48

2.2 Message-passing Methods . . . 49

2.2.1 Centralised Controller . . . 50

2.2.2 Moving Values . . . 52

2.2.3 Odd-even Transposition . . . 53

2.2.4 Fold-over Operation . . . 54

2.3 Shared-memory Approach . . . 56

2.3.1 Parallel Implementation . . . 56

2.3.2 Inertia Detection . . . 57

2.3.3 Experiments . . . 58

2.4 Conclusion . . . 58

The chemical programming model was first proposed in the context of parallel processing as a way of *naturally* expressing parallel computations, encompassing both SIMD and MIMD models. This natural expression of parallelism implies a shift in the way of thinking of the programmer: while in ordinary languages instructions are executed sequentially one after the other in the order laid down by the programmer, there is no need for specific scheduling when executing programs written using the chemical paradigm — instructions are simply executed whenever possible. Hence, it is inherently well suited for execution on multi-processor architectures.

This chapter gives an overview of the research which focused on the execution of chemical programs on such architectures. In order to offer a better understanding of

the methods proposed, we first detail an abstract implementation on a single processor in Section 2.1, and then present the research dealing with executing chemical programs on multiple processors via message-passing, in Section 2.2, and the works adopting the shared-memory approach, in Section 2.3.

2.1 Single-processor Execution

Even though the paradigm of the chemical model is implicitly parallel and non-deterministic, chemical programs have to be executed on real, existing hardware, which is designed with sequential, ordered execution of deterministic steps in mind. The same principle applies to most programming languages, making the mapping of instructions of sequential programs onto machines an automatic and straightforward process, since a sequential command issued in a programming language can be directly translated into one or more processor instructions, rendering any additional scheduling mechanisms superfluous. On the other hand, devising an execution scheme for *chemical* instructions requires some effort: even though the model does not impose any specific scheduling order, it still has to be defined in practice. Here we meet a second challenge: because the chemical programming model offers a higher abstraction than most programming languages, how can its parallel, high-abstraction programs be translated into executable, low-level code? And how to execute this sequential code in parallel?

Algorithm 2.1: Execution of a chemical program.

```
    input  : a multiset M containing m molecules and a rule of arity r
    output: the inert multiset M
 1  while changed = true do
 2      changed ← false;
 3      for x_1 ← 1 to m do
 4          ⋮
 5          for x_r ← 1 to m do
 6              if x_r ∈ [x_1,...,x_{r−1}] then continue;
 7              if cond(M[x_1],...,M[x_r]) = true then
 8                  action(M[x_1],...,M[x_r]);
 9                  remove M[x_1],...,M[x_r] from M;
10                  append the products at the end of M;
11                  changed ← true;
```

We start by laying down the basic idea of executing chemical programs on a single processor. The steps are illustrated in Algorithm 2.1. Given a program in the form of a solution composed of a multiset, M, the algorithm computes the multiset's inert counterpart by sequentially applying the action defined in the rule on combinations of r molecules. For the sake of simplicity, the algorithm assumes that only one rule is present in the solution; extending it to handle multiple rules is straightforward: one more `for – each` loop per rule would have to be added to the algorithm, either as the outermost, or as the innermost one.

While in the chemical model there is no notion of structure or locality, the multiset is here represented as a sequential array or list. Note that, however, there is no dependency in the data ordering — two molecules being *neighbours* in the array does not imply a relationship between them. The algorithm iterates through all of the possible combinations of r molecules (lines 3 — 5) and tries to find those which satisfy the rule's reaction condition. When one such combination has been found (line 7), a reaction is carried out by applying the action specified in the rule's definition (line 8). The reactants are removed from the multiset since they have been consumed in the reaction (line 9), while the products are appended at the end of the multiset (line 10), allowing the newcomers to react with existing molecules in a later iteration of the algorithm.

Each time a reaction takes place, a flag, *changed*, is set, signalling that the computation has to continue because new elements have appeared in the multiset (line 11). Therefore, the execution stops once the execution environment has iterated through all of the combinations of molecules without performing a reaction; inertia has been reached and detected by the environment. Thus, as explained in Section 1.1.3, line 7 has to be executed and evaluated negatively $m!$ times consecutively, depending on the type of the rule present in the program.

To summarise, in order to follow the chemical model's paradigm, the algorithm combines four steps:

- *searching for candidates*: combinations of molecules conforming to the rule's reaction condition are found by permuting the elements of the multiset;
- *atomically capturing reactants*: the paradigm dictates that a molecule can be consumed in at most one reaction, which is in this case implicitly guaranteed since there is only one process executing the program;
- *performing reactions*: the fitting molecules are removed from the multiset, they are consumed by means of the action function, and the products of the reaction are inserted into the multiset;
- *detecting inertia*: by checking all of the possible combinations, the algorithm assures that if inertia has been reached, it is going to be detected.

An execution environment for a single processor has been provided by Radenac [99]. Analysing how these steps can be decoupled and executed in a distributed manner represents the central theme of this thesis. The next sections detail the previous attempts at executing the algorithm in parallel on multiple processors which communicate either through message-passing or by means of a shared memory.

2.2 Message-passing Methods

The algorithms and the methods presented in this section target mostly specific parallel machines, which mainly differ in the way their processors are interconnected. All of them

suppose the execution of a chemical program which contains a single binary rule, *i.e.* a rule of arity 2. Moreover, they assume this rule to be a reducer rule. Finally, each solution allocates only one molecule per available processor. Thus, the size of the multiset matches the number of processors on the machine. When taken together, these three assumptions greatly limit the impact of the approaches presented below as they cannot be extended into a generic runtime.

2.2.1 Centralised Controller

In [11], Banâtre *et al.* give a first idea on how to distribute the execution of a chemical program. They present a synchronous algorithm which uses a centralised controller instructing the processors when to communicate with each other and swap and test values. The processors are connected through a bus in such a way that each processor can communicate with all the others directly. At each step, the controller processor spreads a signal through the network triggering the communications. Three signals are used to guide the computation: S_1, sent by the controller to trigger the communication; S_2, sent by a processor to the controller to signal the occurrence of a reaction; and S_3, sent by a processor to the controller when no reaction took place.

The idea consists in checking all of the possible pairs of molecules by coupling distinct pairs of processors at each step. A processor P_i checks its value against that of processor P_k, where k is initialised to $(n-i)$ **mod** n and incremented (*modulo* n) after each step. After each test, and possible reaction, the controller is informed of the outcome. If $n-1$ consecutive steps without a reaction have passed, it declares inertia and stops the execution.

This synchronous process is shown in Algorithm 2.2. The controller (lines 1 — 9) uses the internal variable t to count the number of steps without a reaction. It sends the signal S_1 to all of the processors to start the step and then waits for their signals. If at least one of them returned S_2 it resets its variable t, otherwise it increments it. This process goes on until t equals n. The computing processors (lines 10 — 33) continuously execute a loop where they *(i)* wait on the controller's signal S_1; *(ii)* communicate with their counterpart processors P_k and test the reaction condition; *(iii)* send the outcome of the test to the controller; and *(iv)* increment their variables k. When P_i communicates with P_k whose index k is less than P_i's index i (lines 14 — 17), it simply sends its value to P_k, waits for a return value and sends the signal S_3 to the controller as it has not performed a reaction. On the other side, the processor receiving the value (lines 18 — 27) checks both combinations of molecules — (v_i, v_k) and (v_k, v_i) — against the reaction condition. If one of the two conditions holds, it performs a reaction, sends one of the newly created molecules back to P_k and S_2 to the controller. Otherwise, S_3 is sent to the controller.

By exploiting symmetry, the algorithm is able to detect inertia in a minimal number of tests. At each step, $^n/_2$ processors do two tests each. Hence, after $n-1$ steps, $n(n-1)$ tests are done. However, this minimality is satisfied only for rules which produce exactly two molecules; if a lesser number is produced, some processors will perform checks in vain using *dummy* molecules due to symmetry.

Algorithm 2.2: Centralised controller.

```
1  on controller
2      t ← 0;
3      while t < n − 1 do
4          send S_1 to all processors P_i;
5          wait for S_i from all P_i;
6          if ∃S_i, S_i = S_2 then
7              t ← 0;
8          else
9              t++;
10 on processor P_i
11     k ← (n − i) mod n;
12     while true do
13         wait for S_1 from controller;
14         if k < i then
15             send v_i to P_k;
16             wait on v_i from P_k;
17             send S_3 to controller;
18         else if k > i then
19             wait on v_k from P_k;
20             if cond(v_i, v_k) then
21                 v_i, v_k ← action(v_i, v_k);
22                 send v_k to P_k;
23                 send S_2 to controller;
24             else if cond(v_k, v_i) then
25                 v_i, v_k ← action(v_k, v_i);
26                 send v_k to P_k;
27                 send S_2 to controller;
28             else
29                 send v_k to P_k;
30                 send S_3 to controller;
31         else
32             send S_3 to controller;
33         k ← (k + 1) mod n;
```

2.2.2 Moving Values

In the same paper [11], Banâtre *et al.* present a second, asynchronous algorithm where no processor has a general view of the system and, thus, each processing element takes local decisions concerning communication and termination. This solution offers a higher level of parallelism and relies on a chain architecture.

Algorithm 2.3: Moving values.

```
1  on processor P_i
2    while n2_i < n − 1 do
3      if d = true then
4        send (v_i, n1_i, n2_i) to P_{i+1};
5        wait on (v_i, n1_i, n2_i) from P_{i+1};
6      else
7        wait on (v_{i−1}, n1_{i−1}, n2_{i−1}) from P_{i−1};
8        if cond(v_i, v_{i−1}) then
9          v_i, v_{i−1} ← action(v_i, v_{i−1});
10         n1_i, n2_i, n1_{i−1}, n2_{i−1} ← 0,0,0,0;
11       else if cond(v_{i−1}, v_i) then
12         v_i, v_{i−1} ← action(v_{i−1}, v_i);
13         n1_i, n2_i, n1_{i−1}, n2_{i−1} ← 0,0,0,0;
14       else
15         n1_i ++,  n1_{i−1} ++;
16         if n1_i, n1_{i−1} ≥ n − 1 then
17           n2_i ++,  n2_{i−1} ++;
18         else
19           n2_i, n2_{i−1} ← 0,0;
20       send (v_i, n1_i, n2_i) to P_{i−1};
21       v_i, n1_i, n2_i ← v_{i−1}, n1_{i−1}, n2_{i−1};
22     if i ∉ [1, n] then d ← not d;
```

The solution consists in spreading the values over the processors, one per processor, denoted P_i, $i \in [1,\dots,n]$. The inner computing processors — $P_2,\dots,P_{n-1}$ — know only their two neighbours, while the border processors — P_1 and P_n — know only one. Each processor is associated a direction, d, and the idea consists in moving each value from one end of the chain to the other until it either reacts with another value or $n-1$ tests have been made with it. The direction d is a boolean indicating to which neighbouring processor to move the value: *true* hands it to P_{i+1}, while *false* moves it to P_{i-1}. Every inner processor chooses its initial direction randomly and it changes it after each value movement. Since P_1 and P_n know only one neighbour each, their directions are fixed to *true* and *false*, respectively.

Each value v_i travels with two associated indicators, n_{1_i} and n_{2_i}; the former indicates the number of exchanges undergone by value v_i, while the latter is the number of consecutive tests done with other values v_k, such that $n_{1_k} \geq n-1$. The exchange of values

continues until each value's indicator n_{2_i} equals to $n-1$.

The code executed by each processor is shown in Algorithm 2.3. When moving a value forward to processor P_{i+1} (lines 3 — 5), P_i sends it the triplet (v_i, n_{1_i}, n_{2_i}), awaits it back and stores it. The actual computation is done by the processor P_i whose d is set to *true* (lines 6 — 21). After receiving the triplet from P_{i-1}, it checks the reaction condition for both combinations. If one holds, a reaction is performed and all of the indicators of both v_i and v_{i-1} are set to 0. Otherwise, their n_1 indicators are incremented and if both of them are greater than $n-1$ or equal to it, the n_2 indicators are also incremented (lines 14 — 19). This step ensures that every value is going to be tested in combination with every other value in the multiset. The actual exchange then takes place, since P_i sends its triplet to P_{i-1} and stores P_{i-1}'s triplet as its own. Finally, all of the inner processors change their directions (line 22) and start another loop iteration.

Inertia is detected in a distributed manner due to asynchronous communications and exchanges of the values' indicators. The algorithm has to do $2n(n-1)$ tests in order to detect it.

The idea behind this algorithm was implemented on an iPSC hypercube with 16 processors — an asynchronous, massively-parallel machine. The algorithm was tested using an implementation of the sieve of Eratosthenes, presented in Section 1.1.1. The authors obtained considerable speed-ups: when executing the program on a single processor, 70 time units were spent, while only around 15 time units were needed to execute it on 16 processors. The highest decrease in time happened in the transition from one to two processors: it took the machine 30 time units to complete the execution on two processors.

2.2.3 Odd-even Transposition

Creveuil [31] and Huang *et al.* [56] both proposed a synchronous algorithm exploiting the idea behind odd-even transposition sorting [67]. In order to examine all of the possible combinations, a processor P_k communicates and exchanges values with its neighbour P_{k+1} depending on its identifier k: on odd-numbered steps, only processors with an odd identifier test and swap combinations with their neighbours, while on even-numbered steps only those with an odd identifier do so. In doing so, multiple comparisons and exchanges take place simultaneously.

The steps taken by the executing machine are presented in Algorithm 2.4. The variable i denotes the step number and j is the number of consecutive steps without a reaction. These two variables are held and manipulated by the controller processor. Variables local to a processor P_k are x_k — the value it holds — and $reaction_k$ — the boolean variable indicating whether a reaction was performed in the previous step.

Firstly, the controller initialises the global variables (line 1) and then instructs the other processors to start a computation cycle (line 3). Each processor P_k executes then the lines 4 — 12. A processor checks if it is its turn to do a computation cycle (line 5). If so, it takes processor P_k's value and it tests both combinations of values against the reaction condition and performs a reaction if one can occur. Finally, it swaps the values x_k and x_{k+1}, in this way concluding a step. The controller checks if at least one processor performed a

Algorithm 2.4: Odd-even transposition.

```
1  i ← 1,  j ← 0;
2  while j < n do
3      for all k ∈ 1,...,n do in parallel
4          reaction_k ← false;
5          if (k + i) mod 2 = 0 then
6              if cond(x_k, x_{k+1}) = true then
7                  x_k, x_{k+1} ← action(x_k, x_{k+1});
8                  reaction_k ← true;
9              else if cond(x_{k+1}, x_k) = true then
10                 x_k, x_{k+1} ← action(x_{k+1}, x_k);
11                 reaction_k ← true;
12             swap x_k and x_{k+1};
13     if ∃reaction_k = true then
14         j ← 0;
15     else
16         j++;
17     i++;
```

reaction and updates the variable j accordingly (lines 13 — 16).

The algorithm needs $n-1$ steps to detect inertia. Due to the odd-even communication pattern, after every two steps all of the processors will have tested two combinations each and exchanged their values with their neighbours. Hence, inertia is detected after $n(n-1)$ comparisons. Creveuil [31] further offers an optimisation for the algorithm for cases in which the rule produces less than two molecules. By applying a parallel data-compaction algorithm, the total number of processors decreases with the size of the multiset, thus eliminating unnecessary tests involving dummy elements.

Creveuil [31] implemented the algorithm on the CM-2 connection machine, while Huang *et al.* [56] implemented it on MasPar MP1, a massively-parallel machine. Both papers report considerable speed-ups and linear scalability in terms of execution time when increasing the number of molecules in the multiset.

2.2.4 Fold-over Operation

Apart from the odd-even transposition, Huang *et al.* [56] present another synchronous algorithm which is based on the fold-over operation [29]. The idea consists in placing the elements on a strip and then folding this strip over from left to right. At each step, the elements residing in the upper segment of the strip are compared in parallel to those in the lower segment. If a reaction happens, the reactants are replaced by dummy elements, while the produced elements are stored in a temporary step and inserted at the right end of the lower segment of the strip after each parallel comparison. The computation continues

until there are no more elements in the lower segment.

Algorithm 2.5: Fold-over operation.

```
1  while is ≠ ie do
2  |   proc[is+1].upper ← proc[is].lower;
3  |   is++;
4  |   if is ≤ iproc and iproc ≤ ih then
5  |   |   if cond(upper, lower) then
6  |   |   |   temp ← action(upper, lower);
7  |   |   |   upper, lower ← ∅;
8  |   |   |   lower[max(ih, ie)+1] ← temp;
9  |   |   |   reset ie;
10 |   |   router[iproc+1].upper ← upper;
11 |   ih++;
12 |   if is+1 ≤ iproc and iproc ≤ ih then
13 |   |   if cond(upper, lower) then
14 |   |   |   temp ← action(upper, lower);
15 |   |   |   upper, lower ← ∅;
16 |   |   |   lower[max(ih, ie)+1] ← temp;
17 |   |   |   reset ie;
18 |   |   router[iproc+1].upper ← upper;
19 |   ih++;
20 collect the data from upper;
```

Algorithm 2.5 shows the computation process. The variables *upper* and *lower* designate the two segments of the strip. Each processor holds one of the elements of the lower strip segment. is and ie are the indices indicating the left and the right end of the strip, respectively, while ih denotes the right end of the upper segment. *iproc* represents the processor's index in the processor array, and *proc* and *router* are global variables allowing a processor to access other processors' local memory.

The first step is to move the leftmost element from the lower segment to the upper one (line 2). Based on its index, a processor then tests the combinations of its part of the lower and upper segments and carries out the reaction, if possible, whose products are inserted at the end of the lower segment (lines 5 — 9). Next, all of the elements in the upper segment are moved one place to the right (line 10). Then, another round of parallel tests and movements is performed (lines 12 — 18). Finally, when there are no more elements in the lower segment, *i.e.* when $is = ie$, the algorithm stops and the result is collected from *upper* (line 20).

It takes $n-1$ loop iterations to complete the algorithm for an inert multiset, since all of the elements in the lower strip segment have to be transferred to the upper one. The number of tests done per loop iteration varies from iteration to iteration and depends on the number of elements in each segment. Concretely, there are $4min(|lower|, |upper|)$

comparisons made each step: $2min(|lower|,|upper|)$ before the first parallel movement of elements and $2min(|lower|,|upper|)$ after it, where $min(|lower|,|upper|)$ denotes the number of elements in the shorter of the two strip segments. As this number increases at each loop iteration from 1 to $^{n}/_{2}$ and then decreases back to 1, by summing up all of the comparisons done, one can observe that inertia is detected after $n(n-1)$ tests have been performed.

The authors implemented the algorithm on the MasPar MP1 massively-parallel machine. The experiments' results show a linear augmentation of the execution time with the problem's size. Moreover, by comparing these results to those obtained for the odd-even transposition, they show that using the fold-over operation completes the execution up to five times faster. Even though both algorithms do the same number of comparisons, the fold-over operation has got a higher level of parallelism.

2.3 Shared-memory Approach

We now present a shared-memory approach to the execution of chemical programs, proposed in [50]. It targets parallel systems in which multiple processors execute their code simultaneously while sharing (partially or completely) the memory space they manipulate. The work presents a more realistic solution compared to the ones described in Section 2.2 since it takes into account the fact that, in practice, there are more molecules in the multiset than there are available processors. Furthermore, the solution is not constrained to the execution of specific types of rules. In the paper, the authors tackle the issues related to the implementation of a chemical runtime on a shared-memory architecture, but also try to reduce the complexity of detecting inertia.

2.3.1 Parallel Implementation

In the basic variant of the implementation, the authors use a straightforward, parallel version of Algorithm 2.1 with a few modifications, presented in Algorithm 2.6. Only the outermost loop is executed in parallel by multiple processors (line 4), since the assumption is that there are only a few processors in the system. All of the variables are shared and accessed through spin locks. Molecules participating in reactions are locked in an atomic fashion using a mutual exclusion algorithm explained below. Additionally, an auxiliary multiset, M', is used, in which reaction products are put. It represents the collection of *next-generation molecules* — the ones which are going to be used in the next iteration of the algorithm. The parallel processes are synchronised using a barrier right before joining the multiset M and its auxiliary counterpart M' (line 14).

In order to lock reaction candidates atomically (line 8), and in this way ensure a molecule cannot be consumed in more than one reaction, the authors propose a variant of the two-phase locking mechanism with deadlock prevention by priorities [17]. Each process can be coloured using three different colours, green, yellow and red, while each molecule is assigned an owner and a pointer pointing to its owner's colour, so that a molecule's colour changes as soon as its owner's colour does.

Algorithm 2.6: Shared-memory implementation.

```
1  while changed = true do
2      changed ← false;
3      M′ ← ∅;
4      for all x_1 ← 1 to m do in parallel
5          ⋮
6          for x_r ← 1 to m do
7              if x_r ∈ [x_1,...,x_{r−1}] then continue;
8              if cond(M[x_1],...,M[x_r]) = true then
9                  if lock(M[x_1],...,M[x_r]) = false then
10                     continue;
11                 M ← M − {M[x_1],...,M[x_r]};
12                 M′ ← M′ ∪ action(M[x_1],...,M[x_r]);
13                 changed ← true;
14     M ← M ∪ M′;
```

Initially, a process′ colour is green. It changes its colour to yellow once it wants to lock the molecules it needs and starts checking whether it can lock each of them. If a molecule's colour is green, *i.e.* it has no owner, the process colours it yellow. If a molecule has got already an owner, then the process with a higher priority is entitled to lock it, while the other waits until either the molecule has been consumed or released by the higher-priority process. After the first locking phase, the process turns its colour to red and checks whether it still owns all of the molecules. If so, it proceeds executing Algorithm 2.6. Otherwise it tries to lock again the molecules which it has lost in the meantime, if they still exist.

2.3.2 Inertia Detection

Clearly, it takes $m!$ tests for the system to detect inertia in the general case, where m is the number of molecules. The authors present three improvements which address the choice of molecules which to test with the aim of reducing the total number of tests carried out.

Element Selection. Assuming the reaction condition is a conjunction of literals, $cond(M[x_1,\ldots,M[x_r]]) = L_1 \wedge \ldots \wedge L_l$, its test can be aborted as soon as one of the literals L_i is evaluated as *false*, since the reaction condition is surely going to be evaluated as *false*. Hence, instead of checking the whole condition in the innermost loop, some literals may be checked sooner, in this way avoiding unnecessary inner-loop iterations. Moreover, due to the commutativity of conjunctions, the literals may be arranged in such a fashion that they are checked as soon as possible. In doing so, certain branches of the search tree may be cut in advance, speeding up the inertia detection process. As an example, consider the simple conjunction $f(x,y) \wedge g(y)$. Here, y should be bounded before x is and

$g(y)$ should be evaluated before $f(x, y)$, since: *(i)* y appears in both literals; and *(ii)* if $g(y)$ is evaluated as *false*, the result of $f(x, y)$ is irrelevant.

Range Restriction. If the literals contain arithmetic and comparison operators, search ranges can be restricted only to molecules satisfying it. Same as with element selection, evaluating such arithmetic literals as soon as possible helps reducing the search space. Moreover, combining these two improvements enables the creation of a constant inertia-detection mechanism, since the number of tests does not depend any more on the number of molecules, but on their values.

Generation Improvement. Following Algorithm 2.6, a check of all of the combinations has to be performed in order to declare inertia. However, since the system keeps track of newly-produced molecules, the algorithm can be modified in a simple way to take into account only these molecules. When starting a new iteration, the auxiliary multiset M' is not emptied, but used in the inner loops. The combinations to be tested (either for literals or for the whole condition) are picked in such a way so that they contain at least one *new* molecule, *i.e.* at least one molecule present in M'. This method further improves the inertia-detection time since it guarantees that combinations which have already been checked will not be checked unnecessarily again.

2.3.3 Experiments

Three programs were used to test the system: the knapsack problem, the shortest path problem and the context-free-grammar parsing problem. Their chemical implementations were run on a shared-memory machine comprised of six processors. The exact details of the configuration have been left out of the paper.

The results suggest the impossibility of executing chemical programs on a shared-memory architecture. Indeed, with the growth of the problem, the execution time drastically rises. Moreover, for most runs the measurements could not be finished within a reasonable amount of time. While this partly happens due to the combinatorial complexity of detecting inertia, the inability to run chemical programs on such a platform also stems from the high level of inter-process synchronisation needed in order to perform reactions.

The aforementioned improvements were also implemented and tested. They exhibit visibly better performance results, as most of the runs were completed. Using the improvements leads to almost-linear speed-ups (about 5.5 for six processors). However, the results also show that they benefit more a single-processor environment than they do a multi-processor one (composed of six processors in this case).

2.4 Conclusion

This chapter introduces the issues to be tackled for executing chemical programs. First, a sequential, single-processor execution algorithm is presented, which exposes the four

main issues a runtime has to tackle in order to provide support for the execution: *(i)* search for reaction candidates; *(ii)* atomic capture of reactants; *(iii)* execution of reactions; and *(iv)* inertia detection. As shown in Section 2.1, a single-processor implementation can implicitly cover all of these steps.

Switching to a multi-processor execution environment brings certain challenges, since both the data and the computation are distributed over multiple processors. To this regard, this chapter discusses the message-passing and the shared-memory approaches for dealing with these issues in Sections 2.2 and 2.3, respectively.

While they provide solutions related to the direct implementation of a runtime for chemical programs, they are rather limited in several ways. Firstly, most of the presented solutions target specific platforms. Even though the algorithms provided and the ideas behind them are interesting, porting them to other architectures and/or systems is a challenge in itself. This is the case of solutions presented in Section 2.2. Furthermore, they are limited to the execution of exclusively binary rules, *i.e.* rules consuming two molecules. In spite of the fact that all of the papers cited in this chapter briefly discuss the possibility of generalising their algorithms, extending them for the execution of general r-ary rules is not an easy and obvious, or in some cases even possible, task. Finally, when applying their solutions to the general case, inertia detection becomes highly inefficient. In Section 2.3 we discussed a shared-memory solution proposing to solve this issue through the analysis of reaction conditions, by decomposing them into literals and testing these literals as soon as possible. While the experimental results show this method as a promising solution, it is not an automated method: the user has to do each individual decomposition for each rule they want to execute. Moreover, when using the as-soon-as-possible execution approach, the execution algorithm itself has to be changed from program to program. Consequently, a direct generalisation of the technique is not feasible.

Part II

Distributed Chemical Computing

Chapter 3

Preliminary Feasibility Study Of Distributed Chemical Platforms

Contents

3.1 **DSM-based Execution Platform** **64**
3.1.1 DSM-inspired Architecture Overview 65
3.1.2 Course of Execution 66
3.1.3 Issues of the DSM-based Platform 67
3.2 **Hierarchical Execution Platform** **68**
3.2.1 Physical Layer Abstraction 69
3.2.2 Execution Flow 69
3.2.3 Condition Checking and Inertia Detection 71
3.2.4 Tree Reorganisation 77
3.3 **Prototype** **80**
3.4 **Evaluation** **81**
3.4.1 Test Programs 81
3.4.2 Results 82
3.5 **Conclusion** **88**

Chapter 2 explored the previous works dealing with the distribution of the execution of chemical programs. While their results are encouraging, the impact of the solutions is rather limited in that:

1. they target only specific platforms;

2. they are constrained to the execution of binary rules;

3. they use the chemical paradigm only for creating a model of the computation, which is then translated into an imperative language and executed on the underlying platform.

This chapter presents our first contribution — a preliminary, feasibility study of constructing a distributed platform for the execution of chemical programs on heterogeneous, large-scale platforms. Our goal is to conceive a platform that can run *any (chemical) program anywhere*. In order to meet it, genericness at two levels is required:

- *platform level* : the execution environment should be able to run chemical programs on top of any platform;

- *program level* : the execution environment should be able to execute any chemical program directly, regardless of its composition and without translation into another language.

To that end, we have studied two approaches based on two commonly-used paradigms in distributed computing: shared memory and message passing. In a first attempt, detailed in Section 3.1, we explore a shared-memory approach in order to distribute the execution of reactions and the inertia detection procedure. However, its inefficiency is quickly established due to the high level of locking needed to keep the consistency. This led us to the investigation of alternative distribution models. The architecture of a hierarchical execution model built atop a peer-to-peer overlay network is thus presented in Section 3.2, where the minimality of its inertia detection mechanism is formally established. We, consequently, built a prototype, outlined in Section 3.3, and conducted experiments. The results concerning the platform's viability and scalability are detailed in Section 3.4.

Note that the work in this chapter concentrates on the distribution of the execution process. Here we assume no knowledge about the program or the data: we consider the chemical engines on nodes to be black boxes which simply report whether combinations of molecules can react, and if so perform the reactions. The aim of this study is to construct a runtime around this black box. Therefore, when we examine the *minimality* of the number of checked combinations, we refer to the minimal number of such tests needed in order to check *all of the possible combinations* of molecules. In other words, the runtime discussed here is a *blind* one, since the diverse properties of the executed programs and their molecules are not taken into account.

3.1 DSM-based Execution Platform

One important part when coming to implement an HOCL runtime is the *scheduler*. The scheduler is the entity which decides which rule should be applied and when. It is responsible for the actual implementation of the non-deterministic execution model. One possible strategy is *round-robin* : every rule is triggered once, each on its turn, until inertia. The

centralised runtime of HOCL presented in [99] deals with this relying on several lists : one containing all of the molecules of the solution, and one containing the rules. Thus, the scheduler is greatly dependent on the management of these lists. We started our study by trying to keep these original ideas while trying to distribute the process. About the solution, this leads to two contradictory objectives :

1. keeping the multiset in an easy-to-access, shared location; and
2. distributing the multiset for the sake of load balancing and bottleneck avoidance

These two objectives seem natural : the first enables the system to quickly search for reactants and access them. The second one is a performance requirement : a shared memory has intrinsic drawbacks regarding performance, as a single memory controller is accessed concurrently by a set of processors. These seemingly contradictory goals have already been pursued together in Distributed Shared Memory (DSM) systems [98], in which programs see a set of distributed memory slots as a single virtual entity to be accessed uniformly.

In the remainder of the section, we are exploring the possibility of using the DSM paradigm to build a distributed runtime for the chemical programming model. The paradigm allows one to conceive a distributed platform while *thinking sequentially*, since the DSM implicitly handles concurrency. In this sense, the DSM paradigm shares some similarities with the chemical programming model, which itself relieves the programmer of handling mutual exclusion, making DSM a natural model to start the study with.

3.1.1 DSM-inspired Architecture Overview

The DSM model provides a logical abstraction of the shared-memory model atop a message-passing distributed system. It combines the best features of the centralised and distributed worlds : each node is equipped with its own local memory, but all these slots are virtually assembled to provide a unique global memory accessed uniformly by programmers. Even though originally conceived for and oriented towards supercomputers and homogeneous clusters of computers, the model has been adapted for various types of systems and is considered nowadays for heterogeneous and large-scale platforms [48, 72, 125]. Thus, using such a model enables one to abstract out the physical network and to tackle the problem at hand — executing a chemical application using multiple processors.

The conceptual, logical view of a distributed runtime based on the DSM model is shown in Figure 3.1. The external application represents any entity which requests the execution of the chemical program it holds. It contacts a node in the DSM-enabled network and transfers it the program to execute. The contacted node will, once the execution finishes, transfer the resulting inert solution to the application.

Every node involved in the execution is equipped with a chemical engine able to apply a reaction rule on a combination of molecules. Therefore, each node participates in the

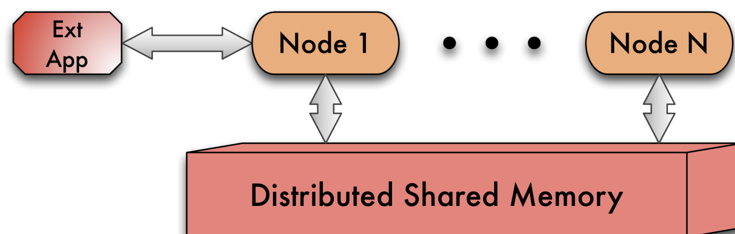


Figure 3.1: Conceptual view of the DSM-based platform.

computation by selecting a combination of molecules, checking it against the rule's reaction condition, and, finally, performing the reaction if the condition is positively evaluated. This process is repeated until inertia, which concludes the computation. However, a logical global space where to access the molecules is not enough to build a chemical runtime. A *scheduler* is needed. We must ensure that each possible combination of molecules is checked once (and only once) against the rules. As we detail in the following, this can be achieved through a list of possible reactant combinations, shared between all nodes involved in the computation.

3.1.2 Course of Execution

We now describe the runtime built around the chemical engines present on the participating nodes. Recall that the runtime devised in this chapter is *blind* with regards to the program being executed: there is no control over molecules nor their properties.

The execution of a program involves four main elements described in the following.

Initialisation. After the transfer of the program to be executed has been completed, the contacted node creates two objects in the DSM: a list containing the rules to be executed and the list of molecules present in the program. As soon as other nodes detect the presence of the list of rules in the system, they each copy it locally and start the execution phase. For the sake of simplicity, let us assume that only one rule consuming two molecules is present in the program. However, the algorithm described is also valid for multiple rules, each with a possibly different number of arguments.

Combination List Filling. The nodes start off by filling the *combination list* — a list containing combinations of molecules which yet have to be tested against the rule's reaction condition. We now briefly describe how the nodes can do it in a cooperative way: each node sequentially picks a molecule, in a round-robin fashion, from the molecule list and creates a sub-list of combinations involving the picked molecule and puts it in the combination list. As an illustration, suppose a program contains five molecules, $m_1,\ldots,m_5$, and is executed on a three-node system. Then, node 1 will put in the combination list the set of combinations containing the molecule m_1 ($\langle m_1, m_2\rangle;\ldots;\langle m_1, m_5\rangle$), node 2 will put the combinations with the molecule m_2 but will avoid the combination with m_1 to

avoid double insertions ($\langle m_2, m_3\rangle; \langle m_2, m_4\rangle; \langle m_2, m_5\rangle$). In the same fashion, node 3 will add ($\langle m_3, m_4\rangle; \langle m_3, m_5\rangle$) to the list.

Condition Checking and Inertia Detection. Once the combination list is in place, each node accesses it, taking the next untested combination of molecules and checking it against the rule's reaction condition. If the molecules cannot react, their combination is removed from the list. Otherwise, the node locks the molecule list and tries to take the molecules from it. If not all of the molecules are available, the combination is deleted from the list and the node simply moves on to the next one. This process is repeated until the combination list has been emptied. When a node notices the list is empty, it repeats the combination list filling process. Inertia has been reached once there are no more combinations to check, *i.e.* when :

1. there are no more elements in the combination list; and
2. the molecule list has been exhausted — there are no more molecules for which combinations can be generated.

The execution is considered to be completed when the state of inertia has been reached. At that point, the node initially contacted by the external application transfers it the inert solution.

Reaction Execution. If, after locking the molecule list, the node is able to obtain all of the molecules it needs to perform a reaction, it removes them from the molecule list and consumes them in the reaction. Then, it removes all of the combinations containing either of the consumed molecules from the combination list. The molecule list is then locked once again. The node fills the combination list with combinations containing the newly created molecules coupled with all of the molecules present in the molecule list, after which the new molecules themselves are added to the molecule list.

3.1.3 Issues of the DSM-based Platform

While a DSM-based approach seems a natural way to implement a distributed chemical platform, it suffers from several issues.

Lists Management. Primarily, the platform relies on the repeated usage of locks over shared objects, not only the molecules, but also the lists used in the computation. The more nodes there are in the system, the more time each of them is likely to spend in a lock's wait queue, thus increasing their idle times and reducing their work times. This is due to the fact that concurrently manipulating a list while keeping its state consistent is inherently a difficult task. Then, the two lists, or *containers*, are needed to coordinate the nodes and keep the system in a consistent state ; locking the list of molecules prevents two nodes from using the same molecule in concurrent reactions, while the list of combinations serves as a *reminder* to nodes as to which combinations yet have to be checked, in

this way preventing nodes to check already checked combinations. The use of lists stems from the DSM model's centralised access to memory — each node has to know *a priori* the memory location (or the object id) it will manipulate. Moreover, each list resides in the memory of one particular node, which entails network traffic penalties during locking, as it has to be transferred from one node to the other.

DSMs in Practice. Finally, on the practical side, implementing such a system would not be an easy task due to the fact that, in spite of the volume of research that has been conducted in the DSM area, there exist only a few DSM implementations, which are tied to specific platforms [24, 58, 74, 108].

In conclusion, while at first glance using a shared-memory approach seems a natural track to be pursued, it suffers from severe drawbacks. Consequently, we shifted our focus towards distributed-memory models. The next section describes a distributed runtime for executing chemical programs with the minimal requirement of message-passing facilities.

3.2 Hierarchical Execution Platform

The proposed platform is illustrated in Figure 3.2. Much like in the proposal based on DSMs, the platform can be seen as a service: the applications submit their chemical programs to be executed and await their respective resulting inert solutions. Note that multiple applications can be processed concurrently by the service. Internally, the platform is composed of three layers detailed in the remainder of the section: *(i)* the overlay network; *(ii)* the execution engine; and *(iii)* the inertia detection mechanism. The overlay network connects the participating nodes and allows the molecules to be spread around the system. The execution engine takes them over and locally performs reactions until the inertia detection mechanism perceives the local solution has reached a stable state, where no more molecules can react.

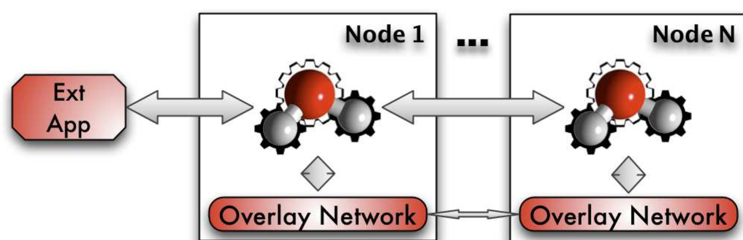


Figure 3.2: The hierarchical platform.

We first explain the role of the underlying overlay network. Next, the functioning of the chemical engine is explained by detailing how molecules are distributed and the computation initiated. Then we focus on the distributed mechanisms required to find reactants and detect the inertia. We show, by providing a first *brute-force* algorithm, that sub-optimality in terms of number of reaction tests can be easily encountered if the algorithm is not designed properly. The optimal algorithm we designed is then given. Its minimality is formally

and experimentally proven. At the end of the section we focus on how to inject load balancing within such an architecture, by presenting a scheme to reorganise the execution flow.

3.2.1 Physical Layer Abstraction

The scale and the heterogeneity of targeted platforms should be abstracted out, before going further in the design of a distributed chemical machine in order to run programs in different environments. This requirement is addressed by relying on overlay networking, which builds logical, virtually-homogeneous networks on top of heterogeneous platforms. One good choice to build an overlay are distributed hash tables (DHTs) [109, 118] in the sense that they partially solve the scalability issue as nodes are guaranteed to communicate efficiently regardless of their number. A DHT's routing complexity typically grows logarithmically with the number of nodes in the platform. Another advantage of DHTs is load-balancing. In particular, the external application sending its program to the chemical runtime platform can contact any of the DHT nodes acting as an entry point to the platform. Then, natural load balancing is obtained as each application can choose a different contact node (through an out-of-band mechanism). The platform completes the requirement by employing the locality principle: computation happens first where data is located.

Any DHT could fill this role. In the following, we use the Pastry DHT [109]. As detailed in Section 1.2.2.3, in Pastry nodes are given a unique identifier chosen uniformly at random in a circular identifier space, organising nodes as a ring. Each node maintains a routing table of shortcuts, allowing systematic routing from any node to any other node in a logarithmic number of hops in the logical network.

3.2.2 Execution Flow

The node contacted by the external application is referred to as the *source* in the remainder. The reception of data triggers the creation of an execution tree rooted at the source for the execution requested by the application. The execution will also finish on the source, which will finally deliver the inert solution, *i.e.*, the result, to the requesting application. We now detail the construction of the execution tree.

Once the source node receives the data of the chemical program, it scatters the data molecules across the Pastry ring according to their hash values; the cryptographic hash function of the underlying DHT guarantees uniform dispersion with high probability (w.h.p.). Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops [109], where n denotes the number of nodes in the platform. In the course of the routing process, the path of each molecule is traced by intermediary nodes, called *forwarders*, from the source node to a molecule's destination node, referred to as a *worker*. By passing on molecules, a forwarder maintains a *local state* (in addition to the Pastry's routing table) containing the set of nodes to which it forwarded molecules. This set of nodes constitutes its child nodes in the execution tree. Note that forwarders, together

with the source node, will be workers as well, w.h.p. Finally, the source node spreads down the tree one final message, *mc* containing the rules to execute.

Upon the receipt of *mc* on a node *p* from a node *s*, *p* completes its state with *s*, referring to it as its *parent* (in the multicast tree being built). In case *p* has already received *mc* from another node, it just drops it, but sends another specific message back to *s*, which, upon receipt, deletes *p* from *s*′ local state. This ensures that, combined, the local states form a tree. This tree, rooted at the source node, will be used later to make partial inert solutions move backwards to the source node. The created tree presents some similarities with the Scribe publish/subscribe system [26]. Note that the complexity of the local state is logarithmic to the number of nodes in the system, as nodes referenced in the local state of a node (except its parent) are necessarily inside Pastry's routing table, itself logarithmic in size.

After receiving the multicast, nodes start locally the computation. Every possible combination of molecules residing on a node is checked on this node against the rules, and, if possible, reactions take place. The combinations′ cardinality is determined by the number of molecules local to the node and the number of a rule's arguments. When the part of the solution received by a node is inert, it must associate itself with other nodes to continue the computation.

Each of them sends its inert local solution to its parent. Parents then add them to their own and continue the computation. Only when a parent has received all of its children's solutions and when its local solution is inert, the process continues with the parent transferring its local solution to its parent, and so forth until all of the inert local solutions reach the source node, which delivers the global solution after executing it until inertia.

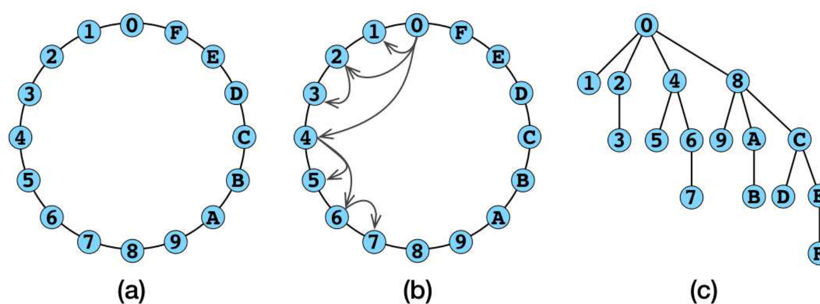


Figure 3.3: Execution example: (a) the original ring; (b) molecule dissemination and tree creation; (c) the execution tree.

Execution Example. Consider the sixteen-node Pastry ring shown on Figure 3.3a. Following Pastry's routing scheme, let node 0's routing table contain nodes 1, 2, 4 and 8, node 1's routing table nodes 2, 3, 5 and 9, and so forth. Furthermore, let node 0 be the source node for a chemical program composed of 16 molecules with IDs 0 through *F*. Node 0 holds on to molecule 0, while sending molecules 1 and 2 to their respective nodes since they are located in its routing table. In doing so, node 0 puts them into its children list in its local state. Molecule 3 is sent to node 2, which forwards it to node 3 and puts it in its

own children list. Next, node 0 sends molecules 4 through 7 to node 4. It keeps molecule 4 and forwards the others to nodes 5 and 6, putting them in its children list. The process of disseminating molecules and creating parent-child relations for the first eight molecules is depicted in Figure 3.3b. Once all of the molecules have reached their destinations, the complete execution tree is in place (Figure 3.3c). It is then used to spread the program's rules and signal the beginning of the execution. In the final phase, local inert solutions travel in the opposite direction — from node F to node E, then to node C, *etc.* Once their local solutions are inert, the nodes will send them upwards to their parents — node F will send its solution to node E, node 7 to node 6, etc. After receiving the inert local solutions from nodes 1, 2, 4 and 8, node 0 starts the last execution cycle and delivers the final result to the requesting application.

Fault Tolerance. While failures can affect our scheme (failures or disconnections of nodes can lead to (i) routing problems, and (ii) loss of molecules), it is not our primary concern here. However, we give a few hints for its reliability. The sub-tree formerly rooted at a crashed node becomes unable to forward its results up the tree. Inspired by the work in [26], a simple detection and reconnection protocol can be defined: when initiating the last multicast message, the source can include its ID. Then, when a node is unable to reach its parent, it can dynamically find a new path to the root by launching a reconnection request in the DHT on the source ID, and thus rebuild the tree. For dealing with the loss of molecules one can rely on state machine replication [78, 112], because each chemical engine operates as a deterministic state machine — it deterministically searches for reactants and performs reactions accordingly. It is thus possible for a node to replicate its state — the molecules currently residing in its local solution — to k neighbouring nodes and periodically send them updates on the progress of its computation. If, after a certain time-out, these neighbours stop receiving updates, the closest one (depending on the topology of the network) includes in its own local solution the molecules it has last received from the failed node. This prevents the loss of molecules, at the same time assuring the continuation of the execution.

3.2.3 Condition Checking and Inertia Detection

Thus far, the focus of this section was on distributing the data and execution of the reactions — the more simultaneous reactions at a time the faster the convergence and thus, the termination.

We now discuss mechanisms enabling *(i)* reaction condition checking and *(ii)* inertia detection. The former can be thought of as being part of the actual execution process — in order to perform a reaction, the runtime has to make sure the reaction condition holds. The latter, on the other hand, is a termination detection mechanism the task of which is to detect the fact that no new reaction can be performed, signalling the execution's completion. Because the execution effectively stands idle while this step is performed, it must be performed as efficiently as possible.

For the sake of discussion, and for an accurate study of the issue, we present two algorithms distributing the task of trying every possible combination of molecules. The first one, based on a repeated brute-force mechanism, is intuitive but sub-optimal, highlighting the fact that, if one is careless, many unnecessary tests can be done, increasing the complexity of an already hard task. The second one, referred to as *BucketSolver*, is shown to detect inertia in a minimal number of steps in terms of the number of combination tests done.

3.2.3.1 Brute-force Algorithm

The approach which seems the most intuitive is that, upon the receipt of molecules from one of its children, a node starts a computation cycle, during which every possible molecule combination of the local solution is tested, possible reactions performed, and thus local inertia reached. If a parent has got g children, there are exactly $g+1$ such cycles. Note that, even though two or more children's results might come at the same time they will not be merged with the node's local solution simultaneously, and will therefore not reduce the number of computation cycles.

Thus, as will the analysis establish later, the total number of tests performed by this algorithm depends on the number of nodes in the tree, the tree's structure and the number of molecules involved in the process. Moreover, this number is higher than the minimal number of combination tests, which limits the system's scalability *vis-à-vis* the number of molecules and the number of participating nodes.

3.2.3.2 BucketSolver Algorithm

As the reader may have noticed, the sub-optimality of the brute-force algorithm comes from its reaction condition checking routine: once a node receives or generates new molecules, it puts them in its unique local solution without keeping track of already checked combinations, leading to future unnecessary tests.

When a node transfers its local solution to its parent, the parent is sure that all of the combinations in the node's local solution have already been tried. The parent does not really need to know exactly which combinations have been checked, as long as it knows the set of molecules they derive from. Thus, in this second algorithm, we create *buckets* into which we put sets of molecules the combinations of which have already been tried.

When a node originally receives molecules from the source, it puts them each in its own bucket. A computation cycle comprises checking only inter-bucket combinations — those whose elements belong to different buckets. For the sake of clarity, let us consider two buckets. Formally, when checking a combination of r arguments, j, $0 < j < r$, elements are picked from bucket a, while $r-j$ elements are picked from bucket b. If the combination is evaluated positively, the elements are removed from their respective buckets and once the reaction has been carried out, each resulting molecule is placed in a new, separate bucket.

Once two initial buckets' intersection combinations have been checked, they are *fused* — the molecules from one bucket are put into the other and the now empty bucket is

deleted. As shown later, the act of fusion guarantees all of the intra- and inter-bucket combinations have been examined. Following this logic, the solution, be it local or global, is declared to be inert once there is only one bucket left on the node or in the system, respectively.

Consider the example illustrated in Figure 3.4. Imagine a node checked two molecules, a_1 and a_2, which now reside in bucket a. The node then receives an inert local solution from one of its children and creates a new bucket, bucket b (Figure 3.4(I)). Now, it checks all of the combinations except those of elements residing in the same bucket. In this example, the node checks the following combinations: (a_1, b_1), (a_1, b_2), (a_2, b_1) and (a_2, b_2) (Figure 3.4(II)). Note that the combinations (a_1, a_2) and (b_1, b_2) are not checked, for they have been previously examined. Finally, presuming no reaction took place, the two buckets are combined into one containing all of the elements (Figure 3.4(III)).

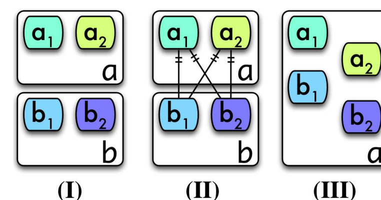


Figure 3.4: (I) Buckets to check, (II) buckets being checked, (III) combined buckets.

This algorithm is valid for an arbitrary number of reactants to find, *i.e.* for an arbitrary size of the left part of rules, since at least one element per bucket must be picked. In presence of one-argument rules the algorithm is adapted as follows. In the case of a reaction rule needing only one molecule to be triggered, then condition checks only need to be performed on the molecules received at the time of the initial dissemination. The ones transmitted from children are only forwarded to parents, as they were already checked against the reaction condition by their initial worker, located in the sub-tree. Thus, every molecule is checked once and only once.

The BucketSolver algorithm provides *inertia detection* (all of the combinations will be examined by the reaction condition) while being *optimal* (every combination will be checked only once). These assertions are proved in Section 3.2.3.3 and corroborated with experimental results in Section 3.4.

3.2.3.3 Inertia Detection Complexity Analysis

In the study of the feasibility of a distributed chemical execution platform, one of the important issues being raised is inertia detection. Here we present the analysis of both algorithms with respect to the total number of combinations examined to detect inertia.

Assumptions and Notation. The number of nodes is denoted by n. The program considered contains m molecules. Thus, after the initial dissemination detailed in Section 3.2.2, every node holds approximately $^m/_n$ molecules. As discussed in Section 1.1.3, we assume, for the sake of simplicity, and without loss of generality, that the program considered now is composed of one commutative rule with r arguments. In other words, unordered combinations of r molecules are checked against the condition of this rule. Under these circum-

stances, we define the optimal number of combinations to be checked as:

$$N_{min} = \binom{m}{r} = \frac{m!}{r!(m-r)!} \tag{3.1}$$

which is the number of tests done by a centralised inertia detection mechanism. Recall, from the discussion in Section 1.1.3, that inertia detection of chemical programs is an NP-complete problem.

In order to analyse and compare the algorithms more easily, we suppose the solution to be already inert, *i.e.* no more reactions can take place. An analysis of an unstable solution is not feasible since the number of reactions, as well as the quantity of deleted and created molecules heavily depends on the rules being executed and the molecules present in the solution. Furthermore, in order to quantitatively compare the two approaches, we assume the execution tree to be a full g-ary tree, where $g \geq 2$. This allows us to analyse and differentiate the approaches more clearly by taking advantage of the symmetry and recursion of full trees. Nevertheless, we prove that the proposed algorithm keeps its efficiency when applied to any type of tree, be it full or not.

Theorem 1. *The number of combinations examined by the brute-force algorithm amounts to:*

$$N = f(m,n,r,g,d) = \binom{m}{r} + n\binom{\frac{m}{n}}{r} + \sum_{i=0}^{d-1}\left[g^{i+1}\binom{m_{c_i}}{r} + g^i\sum_{j=1}^{g-1}\binom{jm_{c_i}+\frac{m}{n}}{r}\right] \tag{3.2}$$

Proof. On a given depth i, $0 \leq i \leq d$, the total number of combinations checked by one node is:

$$N_{i_1} = \binom{\frac{m}{n}}{r} + \sum_{k=1}^{g}\binom{km_{c_i}+\frac{m}{n}}{r} = \sum_{k=0}^{g}\binom{km_{c_i}+\frac{m}{n}}{r} \tag{3.3}$$

where r is the number of arguments needed for one rule and m_{c_i} is the number of molecules received from the child at depth $i+1$ and is defined as:

$$m_{c_i} = \frac{m}{n} * \frac{1}{g-1}\left(g^{d-i}-1\right)$$

Summing by depth all of the checks made, we get the total number of condition checks done by the system N:

$$N = \sum_{i=0}^{d} g^i * N_{i_1} \tag{3.4}$$

After introducing Equation 3.3 and rearranging the terms, Equation 3.4 yields Equation 3.2. □

Following is the quantitative static analysis of the proposed BucketSolver algorithm.

Lemma 1. *When fusing two buckets, a and b, the number of checked combinations totals to:*

$$N_{ab} = \binom{m_a + m_b}{r}$$

where m_a and m_b represent the number of molecules contained in each of the two buckets.

Proof. Given that each of the buckets′ combinations have already been checked, we have to prove their intersection is checked also, *i.e.*

$$N_{a\cap b} = \binom{m_a + m_b}{r} - \binom{m_a}{r} - \binom{m_b}{r}$$

The combinations expressed in the above equation are found by picking r elements from the set $a \cup b$ where j elements come from bucket a and $r - j$ elements from bucket b ($0 < j < r$). When summing the number of combinations, we get the following:

$$\sum_{j=1}^{r-1}\left[\binom{m_a}{j} * \binom{m_b}{r-j}\right] = \binom{m_a + m_b}{r} - \binom{m_a}{r} - \binom{m_b}{r}$$

for $r \geq 2$, which matches $N_{a\cap b}$. □

Corollary 1. *The necessary and sufficient condition that all of the combinations have been checked is having exactly one bucket left.*

Proof. If there is only one bucket left, there are no inter-bucket combinations to be examined, which, by definition, means all of the possible combinations in the present solution have been checked. □

Lemma 2. *The number of combinations checked by BucketSolver in a full g-ary tree amounts to:*

$$N = \binom{m}{r}$$

Proof. In the first computational cycle, every combination will be checked:

$$N_{i_1} = \binom{\frac{m}{n}}{r}$$

Each time a node receives a result from one of its children, it checks $N_{a\cap b}$ combinations, where bucket a represents the molecules already present on the node and bucket b represents the incoming result of a child. A node does so g times, and the number of combinations on it totals to:

$$\begin{aligned} N_{i_1} &= \binom{\frac{m}{n}}{r} + \sum_{k=0}^{g-1}\left[\binom{(k+1)m_{c_i} + \frac{m}{n}}{r} - \binom{km_{c_i} + \frac{m}{n}}{r} - \binom{m_{c_i}}{r}\right] \\ &= \binom{gm_{c_i} + \frac{m}{n}}{r} - g\binom{m_{c_i}}{r} \end{aligned}$$

Knowing that there are g^i nodes on depth i, the total number of checks done can be calculated using Equation 3.4:

$$N = \sum_{i=0}^{d} g^i N_{i_1} = \binom{m}{r}$$

for $g \geq 2$ and $r \geq 2$. □

Lemma 3. *The number of combinations examined by the modified one-argument-rule BucketSolver algorithm is:*

$$N = \binom{m}{1} = m$$

Proof. In the modified algorithm, there is only one computation cycle, which takes place upon the initial molecules dissemination. The number of combinations processed by each node is:

$$N = \binom{\frac{m}{n}}{1} = \frac{m}{n}$$

After this computation cycle, no more computation cycles happen – all of the molecules are being forwarded to the source node. Since there are n nodes participating in the calculation, the number of combinations checked totals to m. □

Corollary 2. *BucketSolver checks every combination only once.*

Proof. It suffices to observe that the number of examined combinations stated in Lemma 2 and Lemma 3 match Equation 3.1. □

Theorem 2. *Lemmas 2 and 3 hold true also for non-regular trees.*

Proof. In the case of a non-regular tree, the number of combinations checked by a node i is:

$$\begin{aligned} N_i = & \binom{\frac{m}{n}}{r} + \sum_{k=0}^{g_i-1} \left[\binom{\frac{m}{n} + \sum_{j=1}^{k+1} m_j}{r} - \binom{\frac{m}{n} + \sum_{j=1}^{k} m_j}{r} - \binom{m_{k+1}}{r} \right] \\ = & \binom{m_i}{r} - \sum_{j=1}^{g_i} \binom{m_j}{r} \end{aligned}$$

where g_i is the number of node i's children, m_j is the number of molecules forwarded to it by its child j and m_i is the final number of molecules which node i will send to its parent:

$$m_i = \frac{m}{n} + \sum_{j=1}^{g_i} m_j$$

m_i will, naturally, become one of node i's parent's m_j, which means they will cancel out one another. Consequently, the total number of combinations checked by the algorithm is:

$$N = \sum_{i=1}^{n} N_i = \binom{m}{r}$$

□

This analysis shows that both presented algorithms insure the detection of inertia. This means that, no matter which one of the algorithms is used in the platform, if a stable state exists, it will be reached.

However, this analysis also shows that the proposed algorithm, BucketSolver, checks every combination only once, and is, thus, optimal. Consequently, it is more efficient than its brute-force counterpart. This indicates its superiority in terms of execution time. Section 3.4 experimentally confirms this statement.

3.2.4 Tree Reorganisation

As explained in Section 3.2.2, the creation of the execution tree heavily depends on the underlying overlay network's properties: its topology (the term *neighbour* having different meanings in different network organisation schemes), its routing algorithm (which directly influences the choice of a node's children in its local state) and also the hash function used in the process (which determines a molecule's final destination). Moreover, given the tree-shaped execution flow, the root node may, depending on the type of problems being solved, face a growing probability of overload, in particular in problems where the number of molecules does not decrease over time. This aspect will be discussed in the evaluation section. Thus, the charge of the tree may be poorly balanced, which could lead to performance degradation and render the algorithms from section 3.2.3 prone to communication and computation bottlenecks with the augmentation of the total number of nodes. To avoid these problems, we are proposing a tree reorganisation scheme, where each group of child nodes is organised into a new sub-tree having the original parent as its root.

3.2.4.1 Reorganisation Scheme

The aim of the reorganisation scheme is to modify the tree's structure in a controlled way in order to minimise congestions and increase the overall performance of the system. The basic idea consists in completely modifying a node's local state, both its children entries as well as its parent entry, which results in a redirection of the node's local inert solution — instead of sending it to the parent *chosen* by the underlying DHT, it will be sent to the parent chosen by the reorganisation scheme. This reorganisation scales gracefully with the system because changes are propagated top-down from the source node and are applied

locally to nodes. Moreover, it is a cost-effective scheme in the sense that no extra messages are needed for said propagation. Rather, the changes are disseminated together with the rules to be executed contained in *mc*, which is expanded to carry additional information concerning the reorganisation scheme: the maximum number of children allowed per node, denoted g_r, and the list of nodes which will be assigned as children to the receiver of *mc*. g_r, $2 \leq g_r < g$, is the reorganisation scheme's input parameter and is set by the platform's user.

The process starts at the source node. The children in its local state are sorted based on their indices and are then split in g_r groups. In each group, the child with the highest index is elected as a *group leader* and remains the source node's child. The rest of the group is removed from the local state and is put into a new *mc* message, which will be sent to the group leader. Once it receives *mc*, the group leader will add the list of nodes present in *mc* to its local state and proceed with the grouping process, and so forth until the changes reach the bottom of the tree. When a node receives an *mc* message, it also completes its local state by indicating that the node which sent it *mc* is its parent. The reorganisation stops once there are no more nodes which need to be reassigned, *i.e.* once all of the nodes have received an *mc* message, which marks the beginning of the execution. The inert solutions will then follow the new paths created while reorganising the execution tree.

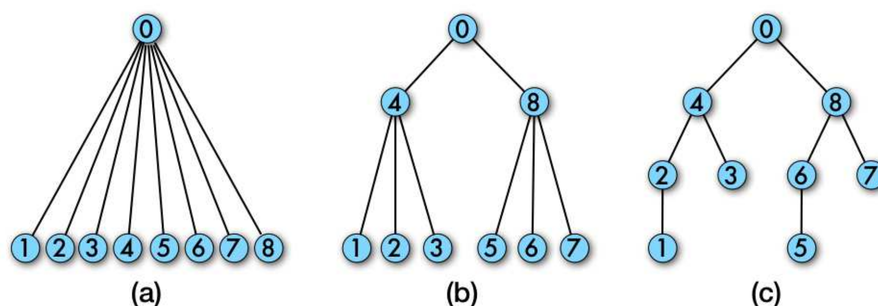


Figure 3.5: Tree reorganisation example: (a) initial state, (b) first level established, (c) tree completely reorganised

Example. Let us demonstrate the reorganisation process on a simple example. Figure 3.5a depicts the execution tree built after disseminating molecules with ids 0 through 8 in a 9-node system. Let node 0 be the source node and let $g_r = 2$. Before sending the message containing the rules, *mc*, node 0 splits its children list in two groups. It keeps node 4 from the first group (1-4) and node 8 from the second (5-8). It then sends one *mc* message including the list of nodes erased from the first group, *i.e.* the list $1,2,3$, to their group leader — node 4. Another *mc*, one containing the list $5,6,7$, is sent to node 8. Node 0 now starts its execution. The receivers of these messages, nodes 4 and 8, update their local states by setting $parent_id = 0$ and copying the lists to their respective children lists (Figure 3.5b). Node 4 now splits its children list into two groups ($< 1,2 >$ and $< 3 >$) and

elects the leaders (2 and 3). Two *mc* messages are then constructed; the first one containing only 1 in its list is sent to node 2, while the other is sent to node 3 without a list. Node 4 now starts the execution. Nodes 2 and 3 set the parent to 4. Node 3 begins executing its local solution, while node 2 sends an *mc* message to node 1 prior to executing the rules. Node 1 now sets its parent and starts the execution. The exact same effect is achieved on the right side of tree: node 8 splits its children list into two groups, sending 5 to node 6 in one *mc* and no list to node 7 in the other (Figure 3.5c). After the reorganisation, instead of all the nodes sending their solutions directly to node 0, node 1 will contact node 2 and node 5 node 6; nodes 2 and 3 will contact node 4 and nodes 6 and 7 node 8. Finally, nodes 4 and 8 will deliver their local solutions to node 0.

3.2.4.2 Analysis

We are now presenting a short analysis of the benefits of workload balancing the reorganisation scheme brings into the system. They are summarised in the following property:

Property 1. *The proposed reorganisation scheme shifts part of the workload from parent nodes to their direct children if the system employs the BucketSolver algorithm.*

Proof. There are two types of workload: the execution of chemical reactions and combination checking, *i.e.* inertia detection. It suffices to prove the correctness of this property only for the latter, since a reaction will happen there where its input combination is first checked. Let us consider an inert solution awaiting inertia detection in a full g-ary tree. On depth $d-1$, the last depth but one, a parent collects the molecules of its g children, each having $^m/_n$ molecules. Following the proof of Lemma 2, the parent will check

$$N_{p_g} = \binom{(g+1)\frac{m}{n}}{r} - g\binom{\frac{m}{n}}{r}$$

combinations in total. If, on the other hand, the reorganisation scheme is activated, the same parent will now have only g_r direct children, and will, thus, check

$$N_{p_r} = \binom{(g+1)\frac{m}{n}}{r} - g_r\binom{\frac{g}{g_r}\frac{m}{n}}{r}$$

combinations of molecules. The comparison of the two expressions yields:

$$g > g_r \Rightarrow g\binom{\frac{m}{n}}{r} < g_r\binom{\frac{g}{g_r}\frac{m}{n}}{r} \Rightarrow N_{p_g} > N_{p_r}$$

Once the g children finish their execution they are no longer part of the tree, so the same principle recursively applies to their parents. Finally, using the logic from Theorem 2, the proof for non-regular trees is trivial. □

It is worth noting here that Property 1 does not entirely hold for the brute-force algorithm. Due to the different formation of local inert solutions induced by the reorganisation scheme, part of the workload of executing chemical reactions will be shifted from parents

to children. However, the amount of checked combinations will not be reduced due to its brute-force search for reactants. On the contrary, the number of checks done by the child nodes elected as group leaders during the reorganisation will increase, effectively increasing the overall number of tested combinations.

3.3 Prototype

To better capture the viability of the platform designed, and thus go further in the feasibility study, a software prototype of our architecture including the algorithms presented before was developed. The sources are available in the `branches/devel-distrib` directory of the *svn* repository located at `http://gforge.inria.fr/scm/?group_id=2125`. Its logical concept is depicted in Figure 3.6.

The prototype exploits FreePastry[1], an implementation developed by the original authors of Pastry, as its overlay network. Its facilities are used by the two units directly above it — the central and the flow unit. These two units represent the implementation of the architecture laid out in Section 3.2.2, while the solver unit implements the reaction condition checking and inertia detection mechanisms from Section 3.2.3.

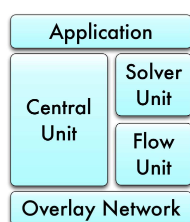


Figure 3.6: Logical concept of the prototype.

Central Unit. The central unit is in charge of accepting and taking over the requests originating from external applications. It hashes the molecules and dispatches them to the underlying overlay network to be routed to the appropriate nodes. It is also in charge of initiating multicasts (such as the one delivering rules to nodes), which it forwards to the flow unit to be sent. When global inertia has been achieved, it delivers the resulting solution to the requesting application.

Flow Unit. Meanwhile, each flow unit monitors the overlay traffic and builds its own local state, *i.e.*, the *localised* view of the execution tree. This localised view is based on routing decisions taken by the overlay network; if a flow unit spots a molecule, coming from node a, being forwarded to node b by the unit's node, it will add node b to the list of its children. Once the dissemination process has been completed, the flow unit holds the list of its direct tree descendants and the parent's identity. This list is used at execution time as a synchronisation barrier: the flow unit forbids sending the local inert solution to the parent until all of its children's results have been received.

Since this unit is the only one directly manipulating a node's local state, it is also in charge of reorganising its part of the tree during the dissemination of multicasts. As described in Section 3.2.4, it elects the group leaders, removes the rest from its children list and passes their identities on to the new group leaders.

[1]`http://www.freepastry.org`

Solver Unit. The key role in the execution is played by the solver unit, which is the implementation of the algorithms proposed in Section 3.2.3. The unit has two different implementations, one for each of the two condition checking methods. After the dissemination process, the central unit triggers its execution of the local solution. Once inert, the solution is transferred to the flow unit. On the other side, when a result from a child is received, the flow unit hands it over to the solver unit, triggering a new execution cycle.

The implementation of the brute-force execution algorithm reflects precisely its description in Section 3.2.3.1, while the proposed algorithm's implementation, outlined in Section 3.2.3.2, carries a slight refinement. The solver unit implements a multi-threaded execution scheme with a minimum amount of synchronisation; the well-defined bucket boundaries imply that two disjoint groups of buckets, each containing more than one bucket, can be executed independently from one another. As a result, the solver unit implements a thread pool which schedules the execution of groups of buckets on free threads in the pool as long as the local solution is not inert, *i.e.,* as long as there are at least two buckets left to fuse. This entails very few synchronisation barriers; a synchronisation point is needed only when *(i)* all of the threads are busy; or *(ii)* there's only one bucket available while there are busy threads.

On the other hand, a similar mechanism cannot be applied to the brute-force algorithm due to the high level of synchronisation required. Indeed, the algorithm has no *recollection* of the combinations it has checked. Instead, it checks sequentially all of the possibilities. As a consequence, access to the combination list ought to be synchronised, which in practice means that most of the threads would waste most of the execution time waiting in the queue. Additionally, access to the molecules themselves has to be synchronised, since two threads might pick combinations containing the same molecule, which hardly constitutes an efficient and scalable solution.

3.4 Evaluation

This section presents an evaluation of the hierarchical platform proposed in Section 3.2. The prototype was tested on two simple chemical programs outlined in Section 3.4.1 after which the results of the experiments are presented.

3.4.1 Test Programs

The evaluation of the proposed architecture and algorithms was conducted using two programs. We consider two families of programs having a different complexity regarding inertia detection. While the expression of programs chosen is quite simple (due to the expressiveness and uncluttered style of the chemical model), their runtime's complexity is similar to many other chemical programs with far more complex logic. Put simply, what matters is the number of test and reactions to be performed over time. The first class of applications has an amount of data that decreases over time, resulting in a decreasing complexity of the combination checking process. The second exhibits a complexity that does not variate during execution.

Programs with Decreasing Complexity. A large collection of real-world applications solve problems the complexity of which gradually decreases as computation progresses. In our experiments we chose to represent this class of applications with the *getmax* program:

let *getmax* =
  replace x, y
  by x
  if $(x \geq y)$

The rule requires two input arguments — two integers —, consumes them, and creates a new molecule which holds the higher value of the two. As noted in Section 1.1.3, this is a so-called *reducer* rule since each reaction it activates decreases the total number of molecules, and thus, the program's complexity. This is a typical scenario for virtually all data-processing applications where the amount of data to be processed diminishes over time. In order to simulate data processing, we introduced a pause of one second after each reaction. In the experiments we performed for this program, a solution containing $50,000$ molecules was used.

Non-decreasing Complexity Programs. The second is a chemical program containing one multiset comprised of 5000 molecules, each composed of two integer numbers — an index and a value associated with it —, and a single rule, *sort*, operating on them:

let *sort* =
  replace $i : x, j : y$
  by $i : y, j : x$
  if $(i < j \,\&\&\, x \geq y)$

The rule consumes two molecules if they are not already sorted in ascending order. Two new molecules are then created, holding the same indices as the original ones, but with swapped values. Although remarkably simple, this program exhibits an important property — it keeps the number of molecules in the solution, as well as the complexity of the program, constant over time, which means that, at the end of the computation, the multiset to be processed by the source node is still large in size, exposing a potential scalability limit of the approach.

Secondly, the rule guarantees the multiset to eventually become inert, allowing us to measure the performance of the algorithms' inertia detection capabilities. This also allows us to compare the two algorithms more precisely.

3.4.2 Results

Experiments were carried out on the prototype described in Section 3.3 with the programs presented above. Their goal is fourfold. We want to: *(i)* examine the feasibility of the

architecture; *(ii)* compare the algorithms from Section 3.2.3; *(iii)* look at the network traffic generated; and *(iv)* test the tree reorganisation scheme laid out in Section 3.2.4.

The experiments were conducted on the French nation-wide Grid'5000[2] [20] computation grid, where the nodes, varying in number from 100 to 1000, were scattered randomly across nine geographically distant sites. The results represent values averaged over 6 runs. The configuration of the logical (DHT) network was changed upon each run.

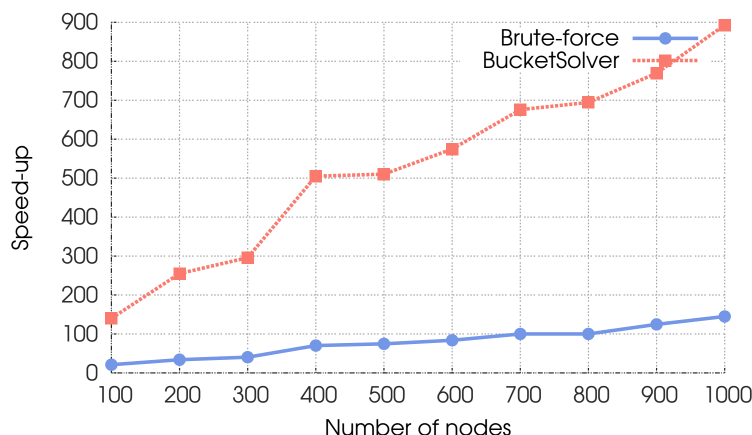


Figure 3.7: Speed-up in execution time for the *getmax* program.

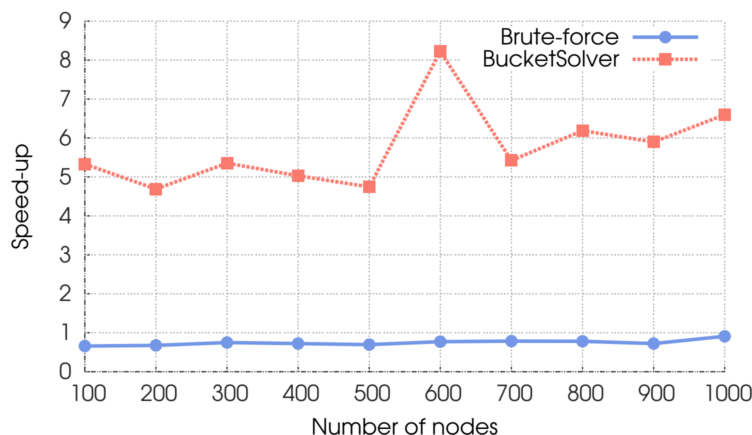


Figure 3.8: Speed-up in execution time for the *sort* program.

Experiment 1 (Execution Time). Firstly, we examine the viability of the framework defined. Figures 3.7 and 3.8 show the speed-up in execution time achieved by the runtime when compared to the execution time of a single runtime instance. This experiment confirms that chemical programs can be directly executed over such a distributed platform, and allows to capture better the architecture's effectiveness, discussed below.

[2] `http://www.grid5000.fr`

When considering the execution time of programs with a decreasing complexity, depicted in Figure 3.7, we observe that using either of the two algorithms results in considerable speed-ups, which linearly grow when increasing the number of nodes. However, there are significant differences in performance between the two algorithms; while with the brute-force approach a maximum speed-up of 150 is achieved, BucketSolver is able to reduce the execution time by a factor of up to 900. Although both algorithms perform the same number of reactions, when using BucketSolver, the runtime benefits from the implementation's optimisation of multi-threading, and thus performs multiple reactions at a time.

On the other hand, Figure 3.8 shows that in the case of the *sort* test program distribution of the execution does not induce a significant speed-up. Concretely, the brute-force algorithm is not even able to match a single instance's execution time regardless of the number of nodes, while BucketSolver achieves speed-ups in the range [5,7]. Note that, due to the non-determinism of the execution and the randomness involved in network construction, the result obtained for 600 nodes, where a speed-up of 8 has been obtained, is in fact an artefact. Still, it is visible that, overall, a global, coherent speed-up is achieved. In contrast to *getmax* where parents in the tree have got less work to do than their child nodes after receiving the results, during the execution of *sort* parents receive the same amount of molecules they initially forwarded to their children. Thus, even though the total number of molecules in the system is constant, the complexity of the program increases over time for each of the participating nodes. As a consequence, parents have got to wait longer on their children's responses, ultimately prolonging the entire execution. Finally, BucketSolver performs better due to its optimality and ability to exploit multi-threading. However, its performance is limited by the fact that the number of nodes involved in the computation is negligible compared to the total number of checks and reactions done during execution, discussed in the next experiment.

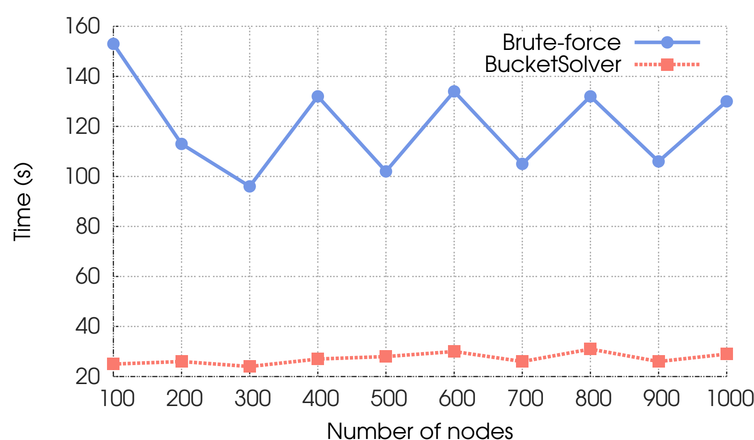


Figure 3.9: Execution time on an inert solution (*sort* program).

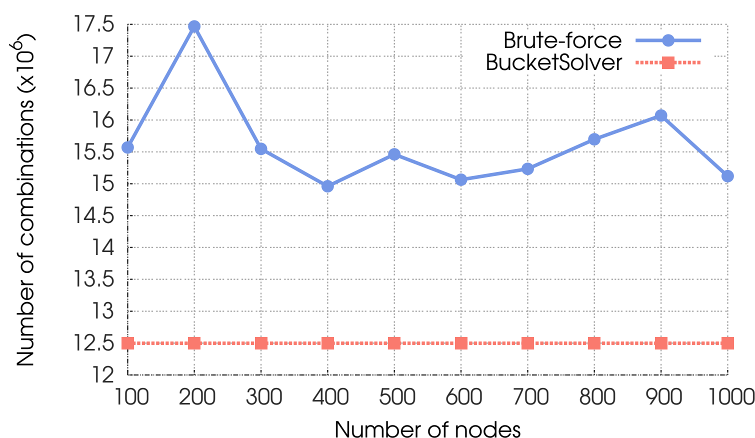


Figure 3.10: Number of checks done on an inert solution (*sort* program).

Experiment 2 (Inertia Detection). Next, we investigate the overhead of distributed inertia detection, depicted on Figures 3.9 and 3.10. The tests were conducted using the *sort* program on an inert solution. The total execution time, depicted in Figure 3.9, shows the brute-force algorithm's inability to efficiently detect inertia in a distributed fashion. Moreover, the fluctuation in execution time reveals its dependence on the structure of the tree built during execution. On the other hand, BucketSolver decreases the inertia detection time four to five times. However, its performance is not improved with the increase in the number of nodes, suggesting that nodes spend a considerable part of their time waiting on results from other nodes. The total number of combinations checked, shown on Figure 3.10, confirms BucketSolver's optimality: the total amount of combinations tested matches that of a single instance regardless of the number of nodes involved in inertia detection. As predicted, this optimality does not hold for the brute-force algorithm, where the number of combinations checked fluctuates wildly.

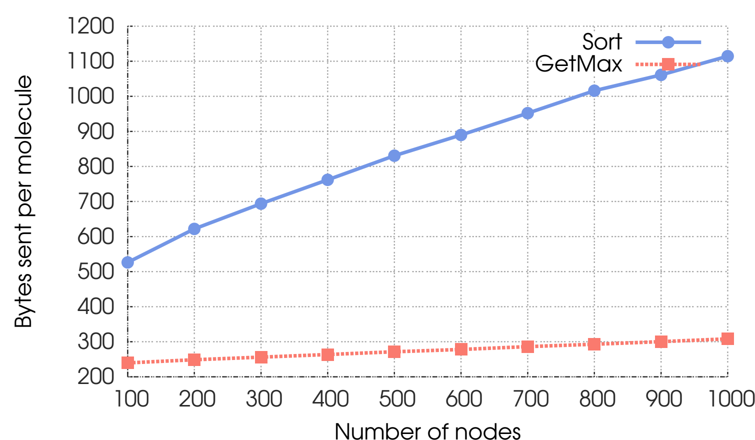


Figure 3.11: Communication costs per molecule for both test programs.

Experiment 3 (Communication Overhead). In this experiment we analyse the scalability related to communication. Depicted in Figure 3.11 is the total number of bytes sent during execution, normalised per molecule. As expected, the overhead of adding new nodes is smaller for *getmax* than for *sort*, since its complexity (and number of molecules) decreases over time. Although steeper, the curve for *sort* shows that distributing the execution has a rather limited impact on network traffic. Moreover, the trend of this overhead is inversely proportional to the growth of the number of nodes: 5 bytes/molecule/node for 100 nodes and 1.1 bytes/molecule/node for 1000 participating nodes. Thus, we can conclude that increasing the number of nodes does not introduce network bottleneck problems.

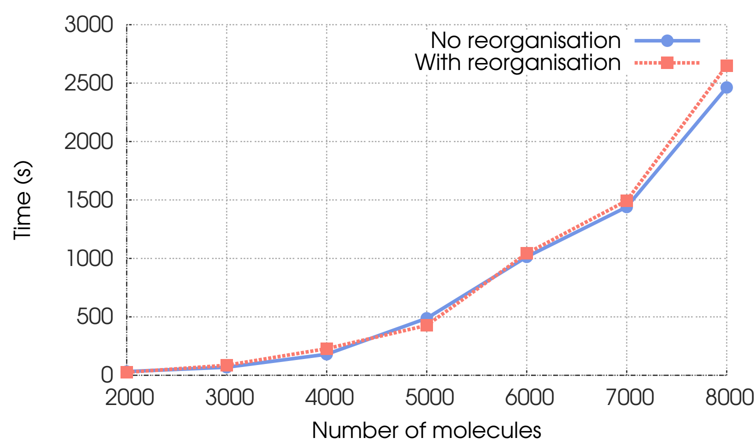


Figure 3.12: Execution time for the *sort* program with and without reorganisation, $n = 600$

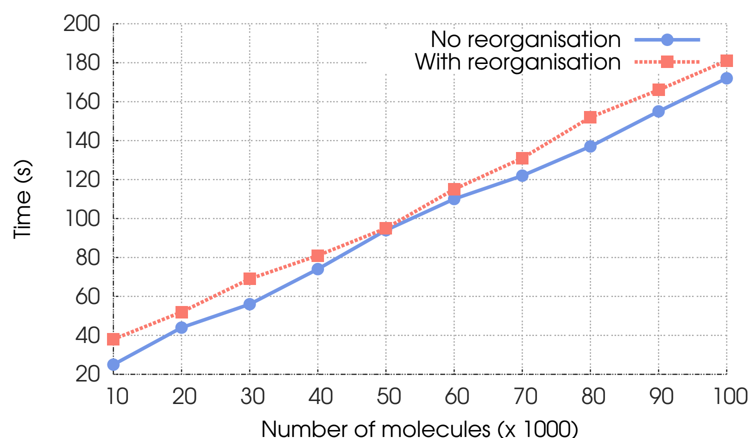


Figure 3.13: Execution time for the *getmax* program with and without reorganisation, $n = 600$

Experiment 4 (Reorganisation Scheme Speed-up). The aim of this experiment is to examine the benefits of tree reorganisation in terms of execution time speed-up. Two sets

of experiments were conducted using the two test programs described in Section 3.4.1. The number of participating nodes was fixed to 600, while varying the size of the problem, *i.e.* the number of molecules. The obtained results are illustrated in Figures 3.12 and 3.13, and show that reorganising the execution tree does not impact much the execution time of the programs. In the case of *sort* (Figure 3.12) there is only a slight variation in execution time, while this difference is greater for *getmax* (Figure 3.13). In both cases there are two factors limiting the execution speed-up of the reorganisation. First, due to the reorganisation, molecules need more time to reach the source node, as their travel paths are longer. The second, and more important, factor is the use of multi-threading. In spite of the fact that, as we show in Section 3.2.4, nodes residing at the same tree depth do less work, this advantage is outperformed by employing multiple threads checking the combinations and executing the reactions.

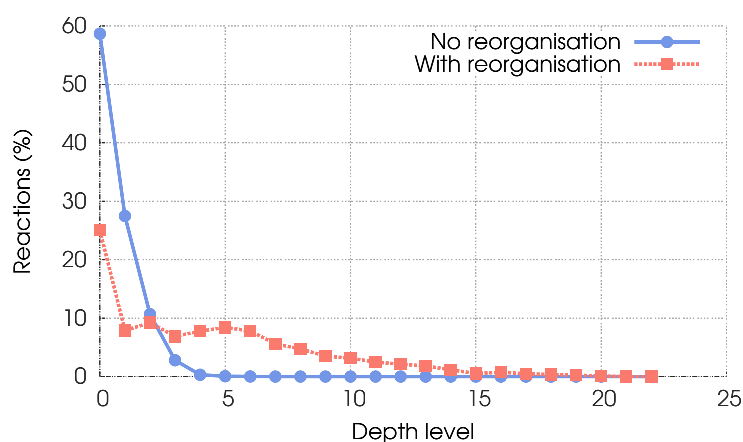


Figure 3.14: Distribution of reactions done per depth for the *sort* program, $n = 600$

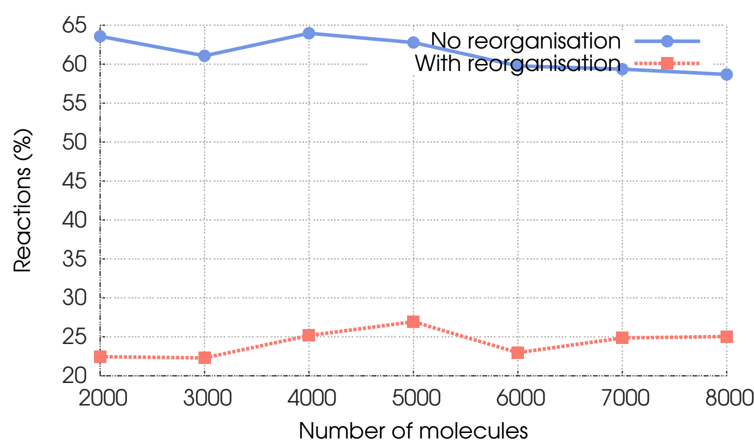


Figure 3.15: Percentage of reactions done by the source node for the *sort* program, $n = 400$

Experiment 5 (Load Balancing). Finally, the theoretical analysis of the reorganisation scheme is put to the test and evaluated using the *sort* program on a varying number of molecules. The results for a 600-node ring are depicted in Figures 3.14 and 3.15 and clearly show that the program exhibits congestion problems on the source node. As shown on these figures, one single node, the source node, can do up to 60% of all of the reactions. Reorganising the tree cuts the percentage by more than a half and augments the height of the tree from 5 to 22 levels, at the same time redistributing the workload down the tree (Figure 3.14). Furthermore, Figure 3.15 reveals that this behaviour persists when changing the size of the problem — in the original version of the algorithm the source node performs from 60% to 65% of reactions, while its counterpart equipped with reorganisation reduces the percentage to 20%–25%. Even though the reorganisation does not have an impact on a program's execution time, it facilitates the load-balancing of the system, which becomes crucial when executing multiple programs simultaneously on a large-scale platform.

3.5 Conclusion

Since previous works considered only limited runtime environments for specific architectures, we started a study of the feasibility of a generic runtime able to execute chemical programs in heterogeneous large-scale systems. We aimed at devising a platform that: *(a)* can automatically distribute and execute any chemical program in the same fashion; and *(b)* is platform-independent both in terms of hardware and software. In this chapter, we discuss two approaches for meeting these criteria : a runtime based on the distributed shared memory model, and a hierarchic one based on peer-to-peer communication.

Our first attempt at a distributed chemical runtime environment exploits the distributed shared memory model. The shared memory space is used to store molecules and rules, as well as a list of combinations of molecules which have to be checked by the nodes against the rules. The management of this list leads to several issues, all of the nodes having to access it continuously in order to progress in the computation, which leads to long wait periods over locks.

For this reason we studied the feasibility of a hierarchical architecture based on message passing in a peer-to-peer network. The design of a tree-structured framework based on distributed hash tables is discussed, and algorithms needed to build a chemical runtime are given, in particular dealing with the crucial inertia detection problem for which an optimal distributed mechanism is provided. The algorithms are formally analysed and the software prototype built and the experimental campaign conducted establish the viability of the concepts presented.

In spite of its viability, performance and low network overhead, the platform suffers from a computation bottleneck on the execution tree's root node. Therefore, we formulated a tree-reorganisation scheme in order to facilitate load balancing and transfer a part of the work deeper in the tree. Even though the results obtained are encouraging, the root node still performs the majority of the work when compared to other nodes, which is a direct consequence of the runtime's hierarchical structuring.

Moreover, the approach developed in this chapter considers the chemical engines to be black boxes which are consulted when a combination of molecules is obtained and needs to be checked against a reaction condition. In other words, the runtime is blind with regards to the actual rules being executed and to the molecules being tested and their properties. In this way, many unnecessary, and in some cases impossible, tests are performed, which prolong the execution.

Taking these circumstances into consideration, the rest of this thesis focuses on: *(i)* fully decentralising the execution process in order to alleviate the problem of the root node's computation bottleneck; and *(ii)* examining the rule's molecule pattern, its reaction condition and the molecules' properties so as to reduce the complexity of the combination-checking routine by eliminating as many unnecessary tests as possible.

Chapter 4

Atomic Capture of Multiple Molecules

Contents

4.1 System Model **93**

4.2 Protocol for the Atomic Capture **94**

4.2.1 Pessimistic Sub-protocol 95

4.2.2 Optimistic Sub-protocol 98

4.2.3 Sub-protocol Mixing 98

4.2.4 Dormant Nodes 100

4.3 Execution of Multiple Rules **101**

4.3.1 Multiple Success Rates 102

4.3.2 Initial Rule Assignment 102

4.3.3 Changing the Active Rule 102

4.3.4 Discussion 103

4.4 Proof of Correctness **104**

4.4.1 Proof of Safety 104

4.4.2 Liveness Proof 104

4.4.3 Convergence Time 106

4.5 Evaluation Set-up **106**

4.6 Experiments Involving One Rule **107**

4.6.1 Execution Time 107

4.6.2 Switch Threshold Impact 108

4.6.3 Switch Behaviour 109

4.6.4 Communication Costs 111
4.7 Experiments with Multiple Rules 113
4.7.1 Multiple-rule Test Programs 113
4.7.2 Execution of the Independent-rules Program 114
4.7.3 Execution of the Dependent-rules Program 116
4.7.4 Execution of the Circular Program 118
4.7.5 Execution of the Workflow Program 118
4.8 Conclusion 122

While the study presented in Chapter 3 shows the viability of a large-scale execution runtime for chemical programs, the hierarchical model proposed is exposed to possible overhead and bottleneck problems on the root node. Since traditional, centralised server-client schemes suffer from similar problems, it is necessary to conceive a decentralised execution runtime for large-scale environments, bringing about new research challenges. One of the most significant is inherent to concurrent rewriting and deals with the atomic capture of the different molecules satisfying a reaction. At run time, a molecule can potentially participate in several concurrent reactions. However, it should be ensured that it will participate in at most one. Otherwise, the logic of the program could be broken.

Let us slightly refine the problem envisioned : we consider a chemical program made of a multiset of objects (molecules), and a set of rules to be applied concurrently on them. Both the objects and the rules are distributed over a set of nodes on which the program runs. Each node periodically tries to fetch molecules needed for the reactions it is trying to perform. As several molecules can satisfy the pattern and conditions of several reactions performed concurrently by different nodes, the same molecule can be requested by several nodes at the same time, inevitably leading to conflicts. Mutual exclusion on the molecules is thus mandatory. Although our problem resembles the classic resource allocation problem [70], it differs from it in several aspects. Firstly, the molecules are interchangeable to some extent. The requested molecules must match a pattern defined in the reaction rule a node wants to perform; if two molecules, A and B, both match the rule's pattern, any of the two may be used in the actual reaction. Then, we differentiate two processes which are :

1. finding molecules matching a pattern (achieved by a *discovery protocol*);

2. obtaining them to perform reactions (achieved by a *capture protocol*).

Consequently, if one node cannot manage to grab some specific molecules, it will switch to another set of molecules. We are not so much interested in avoiding one node's starvation as in the liveness of the system itself : *some* node should be able to perform one reaction in a finite amount of time in order to move the computation forward.

Secondly — and following the previous point — the platform envisioned is at large scale, and the resources dispatched over the nodes are dynamic : molecules are deleted

when they react, and new ones are created. This specific property of the chemical programming model entails that there are no *updates* : a molecule can only be created or deleted, but never updated. Likewise, the number of resources/molecules (and of possible reactions) will fluctuate over time, influencing the design of the capture protocol. Bear in mind that once the holder of a matching molecule is located, the scale of the network is of less importance, since only the nodes requesting the molecules and their holders are involved in the capture protocol.

This chapter proposes an adaptive and efficient distributed protocol for the atomic capture of molecules in large-scale environments. Firstly, in Section 4.1 we describe the system model considered. Then, Section 4.2 details our protocol combining two sub-protocols inspired by previous works on distributed resource allocation but adapted to the distributed runtime of chemical programs. The first sub-protocol, referred to as the *optimistic* one, assumes that the number of molecules satisfying some reaction's pattern and condition is high, so only few conflicts for molecules will arise, nodes being likely to be able to grab distinct sets of molecules. While this protocol is simple, fast, and has a limited communication overhead, it does not ensure liveness when the number of conflicts increases. The second one, called *pessimistic*, slower, and more costly in terms of both execution time and communication, ensures liveness in the presence of an arbitrary number of conflicts. Switching from one protocol to the other is achieved in a scalable, distributed fashion and is based on local success histories in grabbing molecules. Furthermore, we analyse chemical programs containing multiple rules and the possible input/output dependencies they might have and propose in Section 4.3 a rule-changing mechanism instructing nodes as to which rule to execute. A proof of the protocol's correctness is given in Section 4.4. The description of the evaluations conducted is laid out in Section 4.5, while Sections 4.6 and 4.7 discuss the protocol's characteristics through a set of simulation results when one and multiple rules are present in the program, respectively.

4.1 System Model

Different systems require different algorithms for performing atomic operations varying in complexity. We now describe the system model on top of which the protocol is built.

We consider a distributed system $\mathbb{DS}$ consisting of n machines which communicate via message passing. They are interconnected in such a way that a message sent from a node can be delivered, in finite time, to any other node in $\mathbb{DS}$. Moreover, we suppose that a communication channel between any two given nodes is a FIFO queue — a message sent at time t is always delivered strictly before a message sent at time $t+\epsilon$. As we showed in Chapter 3, at large scale such a fully-connected network can be built by relying on P2P systems, more specifically ones employing distributed hash table (DHT) communication protocols [109, 118]. They allow us to focus on the atomic capture of molecules without worrying about the underlying communications' details.

Data and Rules Dissemination. In this chapter, we assume data and rules have already been dispatched to the nodes. In case the data and rules are initially held by a single external application, a dissemination protocol similar to the one described in Chapter 3 can be employed. The external application can contact a node in the DHT and transfer it the chemical solution to be executed. The node which received the data scatters the molecules across the overlay according to the DHT's hash function. Molecules are routed concurrently according to the DHT's routing scheme. The dissemination of rules can follow a similar pattern, or can be broadcast in the network. The only difference is that rules can be replicated on several nodes to satisfy an increased level of parallelism. Throughout this chapter, we simply assume every rule of the program is present on all of the nodes in the system.

Discovery Protocol. In order for the reaction to happen, a suitable combination of molecules has to be found. While the details of this aspect are also abstracted out in the remainder of the chapter, they deserve to be preliminarily discussed. The basic *lookup* mechanism offered by DHTs allows the retrieval of an object according to its (unique) identifier. Unlike the *exact match* functionality provided by DHTs, we require nodes to be able to find *some* molecule satisfying a pattern (*e.g.,* one *integer*) and condition (*e.g., greater than* 3). This can be achieved by the support of range queries on top of the overlay network, *i.e.* mechanisms to find some (at least one) molecules falling within a range, provided the molecules can be totally ordered on a (possibly complex, multi-dimensional) criterion, as for instance provided in [111]. This mechanism can be easily extended to support patterns and conditions involving several molecules. For instance, when trying to capture two molecules ordered in a specific way, a *rule translator* constructs the range query to be sent over the DHT based on the given rule and the first molecule obtained. If matching molecules are found on a given node, this node will trigger the capture protocol.

Fault Tolerance. DHT systems inherently provide fault-tolerance mechanisms. If nodes crash, leave or join, the properties of the communication pattern will be preserved. On top of that, we assume that there exists a higher-level resilience mechanism which prevents the loss of molecules, such as state machine replication [78, 112]. Each node replicates its complete state — the molecules and its current actions — across k neighbouring nodes, where the definition of a *neighbour* depends on the actual DHT scheme used. Thus, in case of its failure, one of its neighbours is able to assume its responsibilities and continue the computation.

4.2 Protocol for the Atomic Capture

Here, the protocol in charge of the atomic capture of molecules is discussed. The protocol can run in two modes, based on two different sub-protocols: an *optimistic* and a *pessimistic* one. The former is a simplified sub-protocol which is employed while the number of possible reactions is high enough to render the possibility of conflicts negligible.

When the ratio between actual and possible reactions drops below a given threshold, the pessimistic sub-protocol is activated. While being the heavier of the two in terms of network traffic, this sub-protocol ensures the liveness of the system, even when an elevated number of nodes in it compete for the same subset of molecules.

4.2.1 Pessimistic Sub-protocol

To some extent similar to the three-phase commit protocol [116], this sub-protocol ensures that at least one node wanting to execute a reaction will succeed. The three-phase commit protocol was originally proposed as a crash recovery protocol for distributed database systems. Its authors study the two-phase protocol and add to it a third, the so-called *prepare commit* phase, thanks to which they are able to obtain a system which is able to abort database transactions in any moment. Although in its essence similar to the three-phase commit protocol, the goal of the pessimistic sub-protocol proposed in this thesis is to secure the liveness of the system by ensuring that at least one node will be able to complete its reaction in a situation where multiple requesters are in conflict over different molecules, as explained below.

Molecule fetching is done in three phases — the *query*, *commitment*, and *fetch* phases — and involves at least two nodes — the node requesting the molecules, called *requester*, and at least one node holding the molecules, called *holder(s)*. Algorithms 4.1 and 4.2 represent the code run on these two entities, respectively, and Figure 4.1 delivers the time diagram of molecule fetching. Note that a node acts at times as a requester (when it executes rules), while at others it behaves as a holder (when it holds a molecule requested by another node).

When molecules suitable for a reaction have been found using the discovery protocol (line 1 in Algorithm 4.1), the query phase begins (line 10). The requester sends *QUERY* messages asynchronously to all of the holders to inform them it is interested in the molecule. Depending on their local states, each of the holders evaluates separately the received message (lines 1—13 in Algorithm 4.2) and replies with one of the following messages:

- *RESP_OK*: the requested molecule is available;
- *RESP_REMOVED*: the requested molecule no longer exists;
- *RESP_TAKEN*: the molecule has already been promised to another node.

Unless it received only *RESP_OK* messages, the requester aborts the fetch and sends *GIVE_UP* messages to holders, informing them it no longer intends to fetch their molecules (line 14 in Algorithm 4.1). Each time a node aborts a grab, it proceeds on to find a new combination of molecules, regardless of the sub-protocol employed or the phase it is in.

Following the query phase is the commitment phase, when the requester tries to secure its position by asking the guarantee from the holders that it will be able to fetch the molecules (line 19 in Algorithm 4.1). It does so using *COMMITMENT* messages. Upon its receipt, each holder sorts all of the requests received during the query phase (line 14 in

Algorithm 4.2) according to the conflict resolution policy (described below). Holders reply, once again, with *RESP_OK*, *RESP_REMOVED* or *RESP_TAKEN* messages. A *RESP_OK* response represents a holder's commitment to deliver its molecule in the last phase. Thus, subsequent *QUERY* and *COMMITMENT* requests from other nodes will be resolved with a *RESP_TAKEN* message. Naturally, if a requester does not receive only *RESP_OK* responses to its *COMMITMENT* requests, it aborts the fetch with *GIVE_UP* messages. The holder then removes the requester from the list, in this way allowing others to fetch the molecule.

Finally, in the fetch phase, the requester issues *FETCH* messages, upon which holders transmit it the requested molecules using *RESP_MOLECULE* messages. From this point on, holders issue *RESP_REMOVED* messages to nodes requesting the molecule.

Conflict Resolution. Each of the holders individually decides to which requester a molecule will be given. Since we want at least one requester to be able to complete its combination of molecules, all holders apply the same conflict resolution scheme, based on the total order of requesters (lines 20—27 in Algorithm 4.2). Any total-order scheme could be applied. We here detail a dynamic scheme based on load-balancing: each of the messages sent by requesters contains two fields — the requester's identifier and the number of reactions it has completed thus far. When two or more requesters are competing for the same molecule, holders give priority to the requester with the lowest number of reactions. In case of a dispute, the requester with a lower node identifier (ensured to be unique by the DHT's hash function) gets the molecule. Such a conflict resolution scheme promotes fairness while at the same time balancing the workload amongst nodes, seeing that the less reactions a node has done the greater the chances are for it to capture the molecules it needs for a reaction.

Discussion. Note that having two phases instead of three is sufficient to ensure the liveness of the system, provided total ordering of nodes is preserved. However, adding a third, *fetch*, phase minimises network traffic. Indeed, with only two phases, this sub-protocol can guarantee liveness is going to be achieved *eventually* since there might be cases where the node which comes first as per the total order does not fetch its molecules immediately, but only after a certain amount of rounds. Consequently, during this period, superfluous network traffic arises since nodes have to return the molecules they have obtained thus far. Consider a simple example of two nodes, N_A and N_B, trying to fetch two molecules, m_1 and m_2, where the priority of N_A is higher than that of N_B. If N_A's *QUERY* request reached the holder of m_1 after N_B's *COMMITMENT* request did, and N_A's *QUERY* request reached the holder of m_2 before N_B's *COMMITMENT*, then each node would obtain only one molecule. Hence, they would be obliged to return these molecules to their respective holders, as neither of the two nodes could complete a reaction, in this way inducing additional network traffic. On the other hand, the three-phase sub-protocol presented here avoids such traffic, since, in the worst case, even if N_A is not able to complete the needed combination, no molecules are sent back and forth, only light messages (*GIVE_UP* in this case).

Algorithm 4.1: Pessimistic Sub-protocol — Requester.

```
1  on event combination found
2      QueryPhase(combination);
3  on event response received
4      if phase = query then
5          QueryPhaseResp(resp_mol);
6      else if phase = commitment then
7          CommitmentPhaseResp(resp_mol);
8      else if phase = fetch then
9          FetchPhaseResp(resp_mol);
10 begin QueryPhase(combination)
11     phase ⇐ query;
12     foreach molecule in combination do
13         dispatch QUERY(molecule);
14 begin QueryPhaseResp(resp_mol)
15     if resp_mol ≠ RESP_OK then
16         Abandon(combination);
17     else if all responses have arrived then
18         CommitmentPhase(combination);
19 begin CommitmentPhase(combination)
20     phase ⇐ commitment;
21     foreach molecule in combination do
22         dispatch COMMITMENT(molecule);
23 begin CommitmentPhaseResp(resp_mol)
24     if resp_mol ≠ RESP_OK then
25         Abandon(combination);
26     else if all responses have arrived then
27         FetchPhase(combination);
28 begin FetchPhase(combination)
29     phase ⇐ fetch;
30     foreach molecule in combination do
31         dispatch FETCH(molecule);
32 begin FetchPhaseResp(resp_mol)
33     add resp_mol to reaction_args;
34     if all responses have arrived then
35         Reaction(reaction_args);
36 begin Abandon(combination)
37     phase ⇐ none;
38     foreach molecule in combination do
39         dispatch GIVE_UP(molecule);
```

Algorithm 4.2: Pessimistic Sub-protocol — Holder.

```
1  on event message received
2      if message = GIVE_UP then
3          remove sender from molecule.list;
4      else if message.molecule does not exist then
5          reply with RESP_REMOVED;
6      else if message = FETCH then
7          clear molecule.list;
8          reply with molecule;
9      else if molecule has a commitment then
10         reply with RESP_TAKEN;
11     else if message = QUERY then
12         add sender to molecule.list;
13         reply with RESP_OK;
14     else if message = COMMITMENT then
15         SortRequesters(molecule);
16         if molecule.locker = sender then
17             reply with RESP_OK;
18         else
19             reply with RESP_TAKEN;
20 begin SortRequesters(molecule)
21     foreach pair of requesters in molecule.list do
22         if req_j.no_r < req_i.no_r then
23             put req_j before req_i;
24             continue;
25         if req_j.id < req_i.id then
26             put req_j before req_i;
27     molecule.locker ⇐ molecule.list (0);
```

4.2.2 Optimistic Sub-protocol

When the probability of successful multiple concurrent reactions is high, the atomic fetch procedure can be relaxed and simplified by adopting a more optimistic approach. The optimistic sub-protocol requires only two phases — the *fetch* and the *notification* phases. Algorithm 4.3 describes the sub-protocol on the requesters' side, while Algorithm 4.4 describes it on the holders' side. The time diagram of the process of obtaining molecules is depicted in Figure 4.2.

Once a node acquires information about suitable candidates, it immediately starts the fetch phase (line 1 in Algorithm 4.3). It dispatches *FETCH* messages to the appropriate holders. As with the pessimistic sub-protocol, the holder can respond using one of the three previously described types of messages (*RESP_MOLECULE*, *RESP_TAKEN* and *RESP_REMOVED*) as shown in Algorithm 4.4. A holder that replied with a *RESP_MOLECULE* message, replies with *RESP_TAKEN* messages to subsequent requests until the requester either returns the molecule or notifies it a reaction took place.

If the requester acquires all of the molecules, the reaction is subsequently performed, and the requester sends out *REACTION* messages to holders to notify them the molecules are being consumed. This causes holders to reply with *RESP_REMOVED* messages to subsequent requests from other requesters. In case the requester received a *RESP_REMOVED* or a *RESP_TAKEN* message, it aborts the reaction and returns the obtained molecules by enclosing them in *GIVE_UP* messages, which allows holders to give them to others.

Conflict Resolution. Given the fact that the optimistic sub-protocol is designed to be executed by nodes in a highly *reactive* stage, there is no need for a strict conflict resolution policy. Instead, the node the request of which first reaches a holder obtains the desired molecule. Consequently, the optimistic sub-protocol is not able to ensure that a reaction will be performed in case of a conflict. In the worst case, all attempts at fetching molecules might be aborted.

4.2.3 Sub-protocol Mixing

During its execution, a program typically passes through two different stages. The first one is the highly reactive stage, which is characterised by a high volume of possible concurrent reactions. In such a scenario, the use of the pessimistic sub-protocol would lead to superfluous network traffic, since the probability of a reaction's success is rather high. Thus, the optimistic approach is enough to deal with concurrent accesses to molecules. The second stage is the quiet stage, when there is a relatively small number of possible reactions. Since this entails highly probable conflicts between nodes, the pessimistic sub-protocol has to be employed in order to ensure the liveness of the system. Thus, the execution environment has to be able to adapt to changes and *switch* to the desired protocol accordingly. Moreover, these protocols have to be able to *coexist* in the same environment, as different nodes may act according to different modalities at the same time.

Algorithm 4.3: Optimistic Sub-protocol — Requester.

```
1  on event combination found
2      foreach molecule in combination do
3          dispatch FETCH(molecule);
4  on event response received
5      if response ≠ RESP_MOLECULE then
6          Abandon(combination);
7          return;
8      add response.molecule to
       reaction_args;
9      if all responses have arrived then
10         NotifyHolders(combination);
11         Reaction(reaction_args);
12 begin NotifyHolders(combination)
13     foreach molecule in combination do
14         dispatch REACTION(molecule);
15 begin Abandon(combination)
16     foreach molecule in combination do
17         dispatch GIVE_UP(molecule);
```

Algorithm 4.4: Optimistic Sub-protocol — Holder.

```
1  on event message received
2      if message = GIVE_UP then
3          molecule.state ⇐ free;
4      else if message = REACTION then
5          remove molecule;
6      else if message.molecule does not
       exist then
7          reply with RESP_REMOVED;
8      else if molecule.state = taken then
9          reply with RESP_TAKEN;
10     else
11         molecule.state ⇐ taken;
12         reply with RESP_MOLECULE;
```

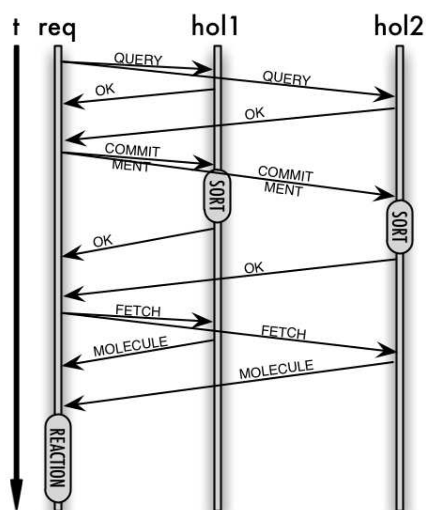


Figure 4.1: Pessimistic exchanges.

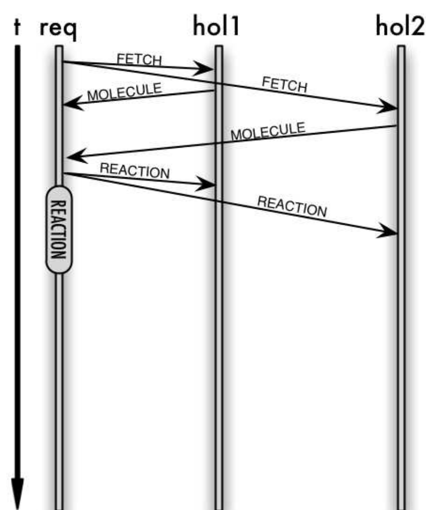


Figure 4.2: Optimistic exchanges.

4.2.3.1 Switching

Ideally, the execution environment should be perceived as a whole in which the switch happens unanimously and simultaneously. Obviously, a global view of the reaction potential cannot be maintained in a large-scale system. Instead, each node independently decides which sub-protocol to employ for each reaction. The decision is first based on a node's local success rate, denoted σ_{local}, computed on the basis of the success history of the last queries the node issued. In order not to base the decision only on its local observations, a node also keeps track of local success rates of other nodes; each time a node receives a request or a reply message, the sender supplies it with its own current history-based success rate, stored into a list (of tunable size). We denote σ the overall success rate, computed as the weighted arithmetic mean of a node's local success rate and the ones collected from other nodes. Finally, the decision as to which protocol to employ depends on the rule a node wishes to execute. More specifically, it is determined by the number of the rule's arguments, since the more molecules the rule needs, the harder it is to assure they will all be obtained. To grab r molecules, a node employs the optimistic sub-protocol if and only if $\sigma^r \geq s$, where r is the number of arguments the chosen rule has and s is a predefined threshold value. If the inequality is not satisfied, the node employs the pessimistic sub-protocol. We show the influence of the switch threshold value on the protocol's performance in Section 4.6.

4.2.3.2 Coexistence

Due to the locality of the switch between sub-protocols, not all participants in the system will perform it in the exact same moment. Nodes may try to grab the same molecules using different sub-protocols. In order to distinguish between *optimistic* and *pessimistic* requests, each requester incorporates a *request type* field into messages. Based on this field, the node holding the conflicting molecule gives priority to nodes employing the more conservative, pessimistic algorithm.

Although this decision discourages *optimistic* nodes and sets them back temporarily, it ensures that a node will be able to grab the molecules it needs *eventually*, since pessimism is favoured over optimism. When pessimistic nodes compete for molecules, one of them is surely going to perform its reaction as per the total order. On the other hand, when solely optimistic nodes compete for molecules, all of their reactions might be aborted since the optimistic sub-protocol does not guarantee the system's liveness. Consequently, pessimistic requests have a higher chance of being concluded by a reaction.

4.2.4 Dormant Nodes

While having more nodes usually yields better performance, this might not be so when executing chemical programs. For instance, during the quiet stage, the system might reach a point where $n \gg m$ (recall that n denotes the number of nodes in the system, while m represents the number of molecules). In this extreme scenario, having more nodes represents a burden for the system, as most of the requests sent for molecules will ultimately

be rejected, elevating the network traffic without speeding up the progression of the computation. Thus we introduce the notion of *dormant pessimistic nodes* — nodes which are using the pessimistic sub-protocol to capture molecules, but do so less often than usual. When a node switches to the pessimistic sub-protocol, it starts counting the number of consecutively aborted reactions. Once this number reaches a threshold a, it puts itself to *sleep* for a predefined amount of time δ — it becomes a *dormant* node. It then *wakes up* and tries to capture another combination of molecules. In case it succeeds, it becomes an active pessimistic node again. Otherwise, it returns to the dormant state for a δ amount of time, and so forth. In order to avoid *massive awakenings* of nodes, *i.e.* the simultaneous resumption of activities of a large number of dormant nodes, we allow the actual amount of time a node spends as dormant to vary by a constant ϵ_δ from δ: before putting itself to sleep, a node randomly chooses the amount of time it is going to sleep for from the interval $[\delta - \epsilon_\delta, \delta + \epsilon_\delta]$. Note that dormant nodes do not put their entire execution on hold — they are still active in the system as molecule holders.

4.3 Execution of Multiple Rules

Thus far we focused on the protocol and its various aspects carrying the assumption that there is only one rule in the program. However, in practice, a small fraction of chemical programs contains only one rule to be used in reactions, as one-rule programs appear rather limited when addressing more complex problems. As every node tries to carry out reactions, when multiple rules are present, each of them has to decide which of the rules it is going to employ in a given cycle. In the remainder of the section, we assume the number of nodes executing the program is greater than the number of rules, *i.e.* $n > n_r$. We refer to the rule being executed by a node at a given moment as its *active rule*.

In order for the computation to be done as smoothly and efficiently as possible, certain constraints of the programming model have to be taken into account. Firstly, a node's decision to switch from one sub-protocol to the other should take into account only the grabs it has tried to do for the rule it is using at the moment. Secondly, due to the paradigm's non-determinism and lack of sequentiality, a rule might be triggered at any given point in time : if a rule is used at time T, but not at time $T + \Delta T_1$, there is no guarantee that it will not be used again at time $T + \Delta T_1 + \Delta T_2$. Finally, the interdependency of rules influences the flow of the execution. Two rules can be:

- *independent* : they can be concurrently executed;
- *dependent* : the product of one rule can be used as input by the other; and
- *circularly dependent* : the product of one rule can be used as input by the other and *vice versa*.

4.3.1 Multiple Success Rates

When deciding which of the sub-protocols to employ to grab molecules, a node should take into account exclusively the capture attempts made while executing the currently active rule, seeing that the execution of a rule does not depend upon the execution of another. Thus, the calculation of the success rate is adapted as follows. Now, each node manages a separate local success grab history list for each rule. Analogously, a separate list of observed remote successes is maintained per rule on each node. Consequently, a node is able to calculate multiple success rates (σ_i), one per rule i, and base its switch decision solely on information relevant to the active rule. Finally, the exchanged messages are expanded with one more field : the identifier of the rule for which the success rate is contained in the message. This way, nodes are able to differentiate success rates for distinct rules and place them in the correct lists. Note that, while the threshold value is set globally for all rules, its interpretation depends on the rule for which the local switch decision is being taken, since a node bases its sub-protocol switch decision only for its currently active rule.

4.3.2 Initial Rule Assignment

To ensure every rule is executed, n_r nodes are each permanently assigned a rule. These nodes are called *rule keepers* and they are selected based on the hash identifiers assigned to the rules : a node N is the rule keeper for a rule R if its node identifier is numerically the closest to the rule's hash identifier. Note that, in case a node, according to the hash function, should be the rule keeper for more than one rule, it may delegate the responsibility for all but one of them to other randomly-selected nodes. Rule keepers try to execute their assigned rules all throughout the computation — they behave as if only one rule (the one they execute) is present in the system. The rest of the nodes ($n - n_r$ of them) pick randomly one of the rules in the program with which to start the execution.

4.3.3 Changing the Active Rule

Even though rule keepers ensure the execution of every rule, the reaction potential of a rule varies throughout the computation; depending on the state of the program at a given moment, more reactants may be present for one rule than another. Thus, nodes ought to be able to change their active rules during the execution based on the reaction potential of the rules. While a node is trying to obtain molecules using the optimistic sub-protocol, a change of the active rule is not being considered, as using this sub-protocol is an indicator of the active rule's high reaction potential. Hence, a change may occur when and only when a node is employing the pessimistic sub-protocol. Given the facts that a rule's reaction potential can be derived from a node's success rate and that every node keeps track of success rates for all of the rules, every node has got a good estimation of the reaction potential of each rule.

A node changes its active rule if the following conditions are met:

1. the node is currently using the pessimistic sub-protocol;
2. the node did not succeed to perform a reaction in the previous cycle; and
3. the success rate for the active rule observed by the node is the smallest success rate when compared to all of the other rules' success rates.

If the above conditions are fulfilled, the node changes its active rule to the one with the highest success rate known to it. If the current σ value of the newly selected active rule permits the node to employ the optimistic protocol, it resets its grab history and sets its σ to 1, since it means that its reaction potential is high, entailing that a fair amount of reactions will be done switching to this rule. Otherwise, it means that the new active rule's reaction potential is also low. In this case the node will, for reasons described in Section 4.2.4, become a dormant node immediately after changing its active rule. Doing so, when it wakes up, it will start trying to apply the new active rule.

4.3.4 Discussion

The reader might have noticed that the presented rule-changing algorithm is a greedy one with respect to both time — *when* to change to another rule — and space — *which* rule to change to. Indeed, by adopting a policy of *late rule changing*, whereby a node changes rules only if its success rate is the worst it knows of, the subset of nodes executing a given rule consumes most of the rule's input molecules. The greediness with respect to space is manifested in the policy to switch to the rule with the highest success rate known to the node about to change rules. While switching to any rule with a value of σ higher than the active rule's would improve the execution, picking the highest one ensures the execution's *optimality*, in the sense that the node choosing it is guaranteed to encounter the least number of conflicts.

Combined, these two levels of greediness assure that:

1. all of the reactions which can be done for a rule will be done, in the sense that if molecules capable of reactivating a rule appear, it is going to be chosen for execution, *i.e.* no reaction is going to be *forgotten* as a consequence of rule switching; and that
2. nodes avoid conflicts as much as possible, seeing that the rule with the highest known value of σ is chosen.

It is worth pointing out that, due to the fact that there is no global view of the system as nodes build up their knowledge based on the communication with other nodes and that they choose a rule at random during initialisation, nodes will decide to execute different rules at distinct times. Moreover, they will not uniformly choose the same rule.

4.4 Proof of Correctness

To be *correct*, this protocol must guarantee two properties:

- *safety*: a molecule is used in at most one reaction (as we consider that every reaction consumes all of the molecules entering it); and
- *liveness*: if there are molecules able to produce reactions in the system, reactions are going to be carried out.

4.4.1 Proof of Safety

Even though multiple *servers* (molecules holders) and multiple *clients* (molecule requesters) are involved in the process, safety is straightforward to prove, because both sub-protocols are synchronised by molecule requesters.

Theorem 3. *A molecule is consumed in at most one reaction.*

Proof. As visible in Algorithms 4.1 and 4.3, before performing reactions requesters wait for the responses of all concerned molecule holders. A reaction is carried out if and only if all of the molecules are present on the requester. Does it not receive a molecule, a requester renounces performing the reaction by executing the Abandon routine, giving back all of the molecules it has captured. Additionally, when employing the pessimistic sub-protocol, a requester has to pass through three synchronisation barriers, one after each phase.

On the other side, each holder locally enforces conflict resolution. Because both sub-protocols have conflict resolution policies which ensure that a molecule can be given to only one requester, a molecule will be consumed in at most one reaction. This is observable in Algorithms 4.2 and 4.4, where a molecule's state changes based on the arrived request. Once it has been promised or given to a requester, others receive either a *RESP_TAKEN* or a *RESP_REMOVED* message even if the molecule can be given back later to the holder due to a requester's call to the Abandon routine.

Finally, there are two cases of conflict between the two protocols. When an optimistic request arrives before a pessimistic one, the pessimistic request is aborted because the molecule has already been reserved by the optimistic requester. On the other hand, if a pessimistic request arrives first, the optimistic request is aborted in favour of the pessimistic one. □

4.4.2 Liveness Proof

To prove the liveness property, we show that:

- the protocol is deadlock-free;
- if no successful reaction happens in the system, nodes eventually switch to the pessimistic protocol;

- if several pessimistic requesters are in conflict, at least one reaction is not aborted;

In addition, we show that a node cannot see its reactions infinitely aborted, *i.e.* that the protocol is starvation-free.

Lemma 4. *A node's execution cannot be blocked infinitely.*

Proof. Although requesters compete against each other for molecules, ultimately the decision is taken unilaterally by the holders on which the molecules reside. This decision is communicated as a response to each request. Due to the usage of reliable FIFO channels, a requester will always get a response for each sent request. Based on the received responses, it will either perform the reaction or abort it and continue its execution. □

Lemma 5. *If an optimistic node sees its reactions continuously aborted, it eventually switches to the pessimistic sub-protocol.*

Proof. When a request of a node is aborted, the node decreases its value of σ (see Section 4.2.3). With each message sent, a node includes the information about its local σ, and collects the values received from other nodes. If there are many conflicts during a certain period of time, all the more so if there is no successful reaction, the local values of σ of all of the nodes decrease. This effect leads to a situation where the computed value of σ^r for all new reactions is lower than the threshold s, which forces nodes to use the pessimistic protocol upon the initiation of new requests. □

Lemma 6. *Eventually, at least one node will succeed in performing a reaction.*

Proof. Initially, and hopefully most of the time, nodes use the optimistic sub-protocol for their requests. Nevertheless, in case of a conflict between two optimistic requesters, both requests can easily be aborted. Consider the example where two concurrent requesters try to capture two molecules, A and B. If the first requester succeeds in grabbing A while the second captures B, then the two requests will be aborted. If such scenarios persist, as per Lemma 5, nodes will switch to the pessimistic sub-protocol.

For the pessimistic sub-protocol, we define a total order based on the number of successfully completed reactions by a node and its unique id. In case of a conflict, all of the reactions might be aborted except for one — the reaction initiated by the node which comes first as per the total order. Since that node has got the highest priority system-wide, all of the holders it contacts will decide in its favour. □

Following Lemmas 4—6 we have:

Theorem 4. *The protocol assures the system's liveness property holds.*

For the sake of completeness, we now prove that the protocol is starvation-free.

Lemma 7. *A node cannot see its reactions infinitely aborted.*

Proof. There are two possible outcome scenarios when a node enters in a conflict over molecules: *(i)* the molecules exist long enough for a node to capture them; and *(ii)* the molecules are taken by another node. Following Lemma 5, a node trying to obtain molecules will eventually switch to the pessimistic sub-protocol.

Because the total order is based on the number of successful reactions, if the node, in case of an abort, tries again infinitely to request molecules for its reaction, eventually, provided the requested molecules are still available, the reaction will take place, given the fact that its position moves up the total order when other nodes succeed in executing their reactions.

If, however, the node does not have the highest priority amongst the nodes in conflict for the molecules, another node will grab them, in this way raising other nodes' positions up the total order. The original node will then try to grab another combination of molecules. It will change the combination until it becomes the node with the least number of reactions performed, at which point it will have the highest priority in the total order. □

4.4.3 Convergence Time

When presenting algorithms for atomic capture, it is customary to study their convergence times. However, any discussion about convergence when dealing with the chemical programming model is not feasible, as convergence itself, and thus the convergence time, is an application-specific property. However, in the remainder of the chapter we present an evaluation of the proposed algorithm, which sheds some light on the subject.

4.5 Evaluation Set-up

Our protocol was simulated in order to better capture its performance. We developed a Python-based, discrete-time simulator, including a DHT layer performing the random dissemination of a set of molecules over the nodes, on top of which the layer containing the capture protocol itself was built. At this layer, any message issued at step t will be received and processed by the destination node at time $t+1$. Moreover, each time a capture attempt either led to a reaction or to an abort, the node tries to fetch another set of r randomly chosen molecules, where r depends on the program being simulated atop the protocol.

Unless otherwise noted, all presented experiments simulate a system of 250 nodes trying to execute a chemical program containing a solution with 15000 molecules. The reactions' durations are assumed negligible, as this allows us to concentrate exclusively on evaluating the capture protocol itself, without having to deal with application-specific problems. For all of the simulations, we used the following constants:

- $a = 10$ (number of consecutive aborts)
- $\delta = 20$ (sleep interval)
- $\epsilon_\delta = 4$ (sleep interval deviation)

- $s = 0.7$ (switch threshold)

Each simulation was run 50 times and the figures presented below show the values obtained by averaging result data from these runs. As the deviation for each simulation is negligible, we here present only the averaged values.

There are two sets of experiments. In the first one we extensively tested the protocol's behaviour, its performance and the network traffic generated by a simple program with a single rule. The second set of experiments examines the system's behaviour when faced with the execution of programs with multiple rules.

For the purpose of evaluating the protocol and its characteristics we simulated five different programs on top of the protocol itself. Note that the programs do not represent concrete implementations of applications since the actual reactions and their results are not taken into account. Rather, they were conceived in such a way as to examine the protocol from different perspectives. The programs are designed to ensure inertia will be reached in a finite number of steps. The problem of distributed inertia detection is currently left aside, but it is going to be addressed in Chapter 5. These five programs cover all of the rule-dependency patterns described in Section 4.3, and thus provide a complete insight into the protocol's characteristics. The next section presents the results obtained by simulating the single-rule program, while Section 4.7 describes the multiple-rule programs and the outcomes of their experimentations.

4.6 Experiments Involving One Rule

The first program simulated is a simple one consisting only of a straightforward rule which simply consumes two molecules without producing new ones. Having only one rule in the solution allows us to concentrate and analyse solely the protocol, its sub-protocols and the switch between them.

4.6.1 Execution Time

Firstly we evaluate separately the performance characteristics of both sub-protocols. Figure 4.3 shows the averaged number of reactions left to execute at each step, until inertia, using only the optimistic mode, only the pessimistic mode, and the complete protocol with switches between sub-protocols with and without the optimisation of dormant nodes, respectively. The *theoretic optimum* curve represents the amount of steps needed to complete the execution in a centralised system, as it represents a system in which the highest possible parallelism degree is possible with regards to the protocol's execution and no conflicts arise during the whole computation. Considering that we need at least two steps to fetch molecules (one to request them and one to receive them), such a centralised system needs $2 * \frac{m}{nr}$ steps to conclude the computation. This represents a lower bound on the number of steps, regardless of the model of computation, be it chemical or other. Note that a logarithmic scale is used for the number of reactions left. The figure shows that, when using only the *optimistic* sub-protocol, there is a strong decline in the number of

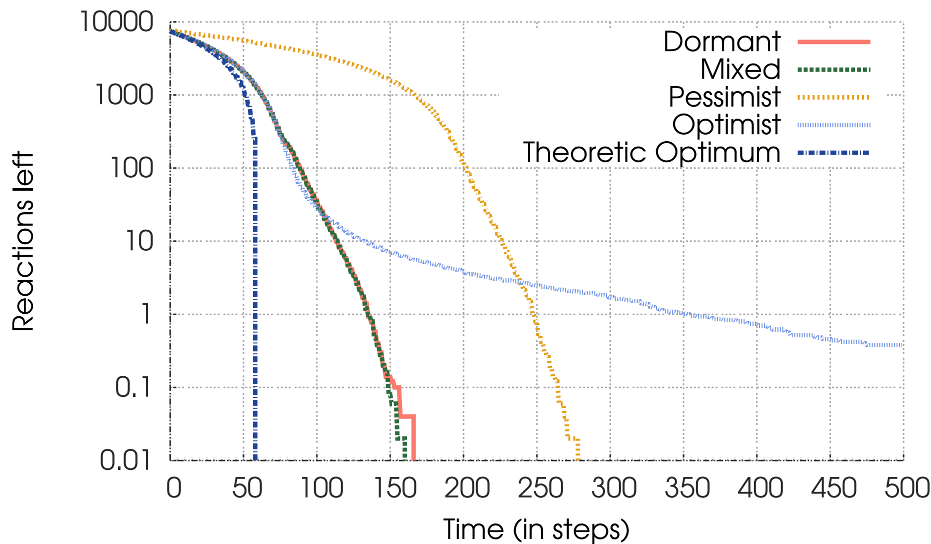


Figure 4.3: Performance comparison of the protocol's variants.

reactions left at the beginning of the computation, *i.e.* when a lot of reactions are possible, and that, thus, there are only few conflicts in the requests. However, it gets harder for nodes to grab molecules when this number declines. In fact, the system is not even able, for most of the runs, to finish the execution, as the few reactions left are never executed, constantly generating conflicts at fetch time. When the nodes are all *pessimistic*, there is a steady, linear decrease in the number of reactions left, and the system is able to reach inertia in a reasonable amount of time, thanks to the liveness ensured in this mode. For most steps, the *mixed* curve traces the exact same path as the *optimistic* one, which means that during this period the nodes employ the optimistic sub-protocol. However, at the end, the system is able to quickly finish the execution as an aftermath of switching to the pessimistic sub-protocol. After the switch, it diverges from the optimistic one to mimic the pessimistic curve, exhibiting a 42% performance boost compared to the performance of the pessimistic sub-protocol. Comparing the theoretic optimum to our protocol, we notice an increase of 166% in the number of steps needed to reach inertia. This is understandable, because there is usually a coordinator in centralised systems with which conflict situations can be circumvented, but it opens the door to serious defaults, such as single-point-of-failure or bottleneck problems. Finally, as far as performance is concerned, including dormant nodes leads to similar results.

4.6.2 Switch Threshold Impact

Next, we want to assess the impact of s (the switch threshold) on the overall performance of the system. Figure 4.4 depicts, in the same logarithmic scale, the number of reactions left after each step for different threshold values, varying from 0.1 to 0.9. As suspected, the curves overlap during most steps, most nodes employing the optimistic sub-protocol. The first curve to diverge is the one where the switch threshold is set very high, to $s = 0.9$.

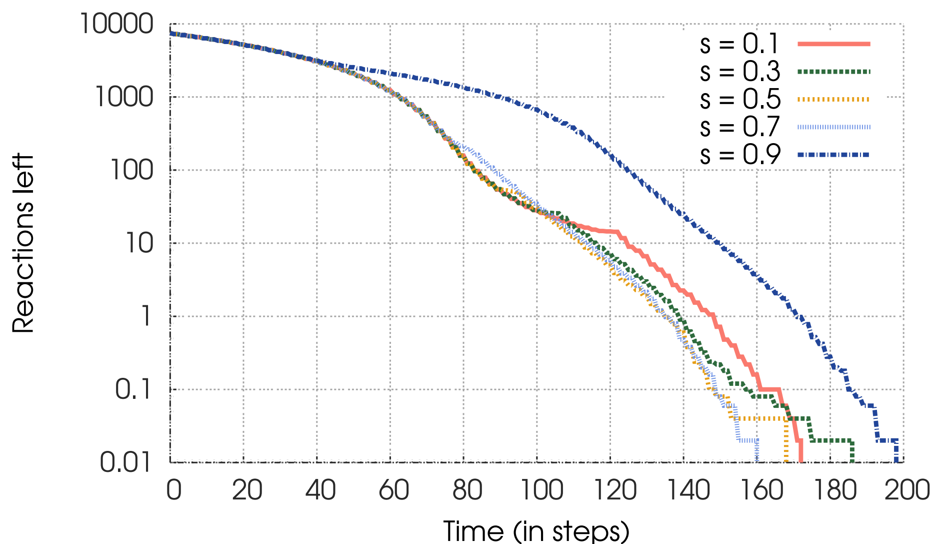


Figure 4.4: Execution time for different switch thresholds.

Because the system depicted by that curve did not fully exploit the optimistic sub-protocol, it is the last to finish the execution. Although slightly, the other curves start diverging at different moments, and, thus, complete the execution after a different number of steps. Figure 4.4 shows that, out of the five values tested for the switch threshold, $s = 0.7$ yields the best performance results in this particular scenario. Looking at completion times for different switch threshold values brings us to the conclusion that the switch threshold can have a significant impact on performance; in this case the execution time can be decreased by up to 20%. Since the performance depends not only on the switch threshold but also on the application being executed, finding an optimal value for s in the general case falls out of the scope of this thesis.

4.6.3 Switch Behaviour

Here we examine the properties of the process of switching from one sub-protocol to the other, represented in Figures 4.5 and 4.6. Figure 4.5 depicts the evolution of the number of nodes in each mode during the execution. We can see that, at the beginning of the execution, all of the nodes start grabbing molecules by using the optimistic sub-protocol. The switch happens about half way through the execution. Around that time, *optimistic* nodes start aborting more and more reactions, and thus can no longer efficiently capture molecules, so they switch to the pessimistic sub-protocol. We observe that, thanks to the systematic exchanges of local σ values, nodes in the system reach a global consensus rather quickly — for a system with 250 nodes, at most 15 steps are needed for all of the nodes to switch to the pessimistic sub-protocol. In other words, the complete transition from using the optimistic sub-protocol to using the pessimistic one constitutes at most 10% of the execution time. For the following 15 steps all of the nodes are active – and pessimistic – and try to capture molecules. However, as the concentration of molecules

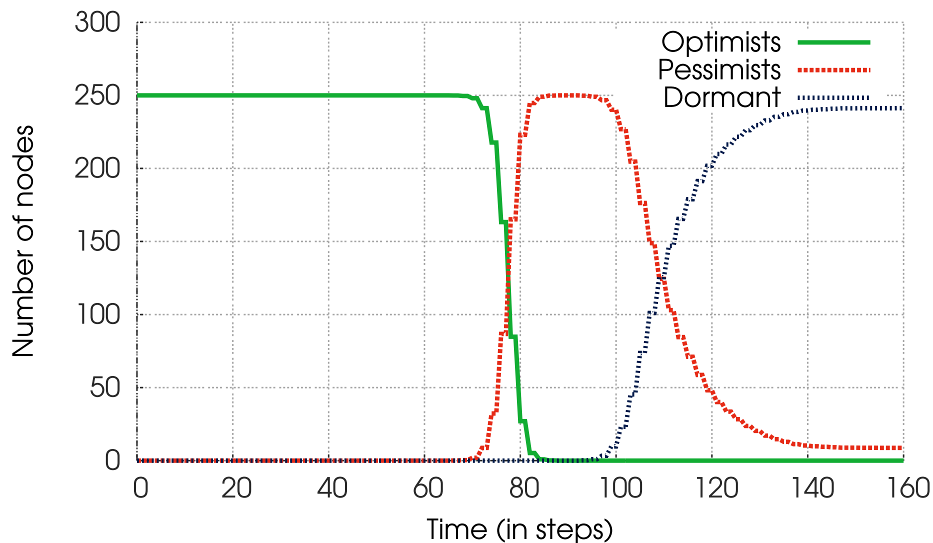


Figure 4.5: Nodes employing optimistic and pessimistic sub-protocols and dormant nodes per step.

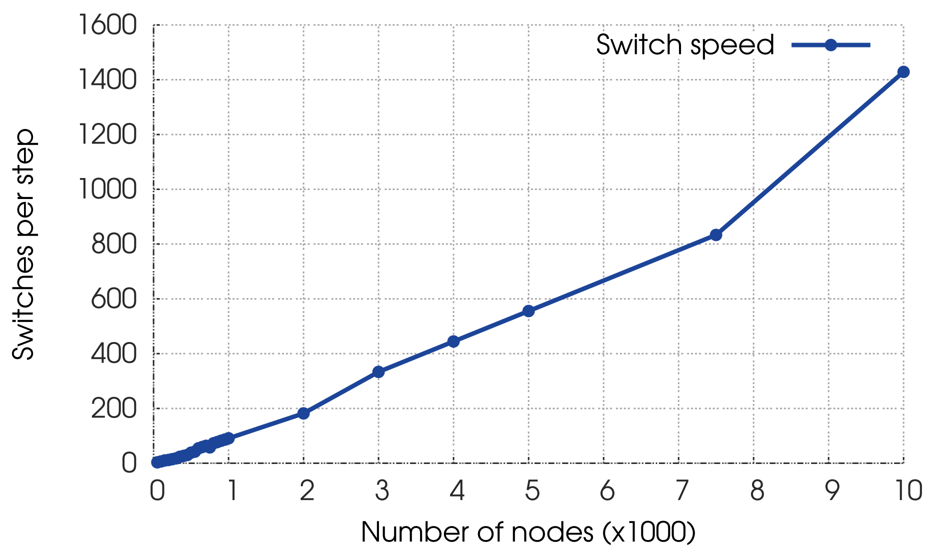


Figure 4.6: Switch speed expressed as number of switches per step (for a constant number of molecules).

further drops, the number of dormant nodes increases, in this way reducing network traffic and allowing the still active nodes to capture the wanted molecules with more ease. Note that, while the overall number of dormant nodes increases, nodes wake up after a certain period and become pessimistic again. Still, one can observe that, overall, there are more nodes asleep than pessimistic ones.

Figure 4.6 illustrates the number of nodes that switch from the optimistic to the pessimistic sub-protocol on each step during the transition period. One can observe that the more nodes in the system, the greater the number of nodes that switch per step. This behaviour comes from the fact that an increase in the number of nodes implies a greater accuracy of the system's state estimation σ as each node communicates with a wider spectre of nodes. As a consequence of this increased accuracy, the number of switches per steps grows quicker and quicker. This shows that the system, regardless of its size, can react quickly to changes, even though there is no global view of the situation.

As discussed in Section 4.2.3, pessimistic requests are favoured over optimistic ones. However, we also conducted simulations when the inverse is true, *i.e.* when optimistic requests are favoured, in order to verify that this design choice does not impact negatively the protocol's performance. The results obtained are similar to those shown in Figure 4.5: there is no difference in the total execution time nor in the switch speed. The figure is, therefore, omitted. The similarities stem from *(i)* the quick propagation of local σ values; and *(ii)* the pessimistic requests' higher chance of completing a capture cycle. Indeed, even though optimistic requests are favoured, as the concentration of available molecules in the system drops, it gets harder for nodes employing the optimistic sub-protocol to capture all of the molecules needed for their reactions.

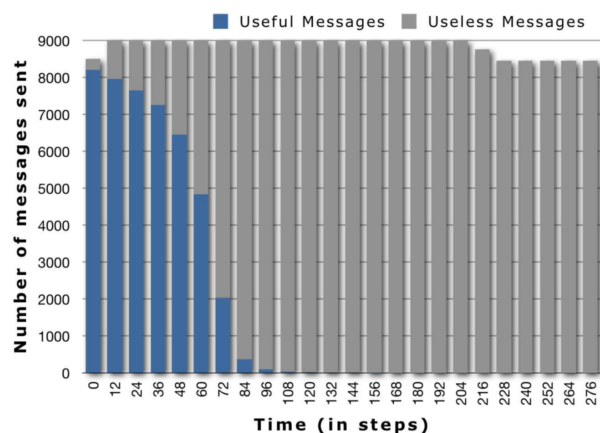


Figure 4.7: Generated messages when only the optimistic sub-protocol is active.

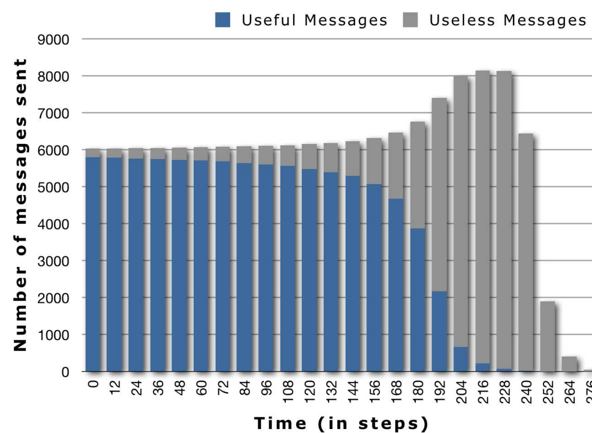


Figure 4.8: Generated messages when only the pessimistic sub-protocol is active.

4.6.4 Communication Costs

Next, we investigate the communication costs involved in the process. Figures 4.7, 4.8, 4.9 and 4.10 depict the number of messages sent per cycle in a system of 250 nodes. One

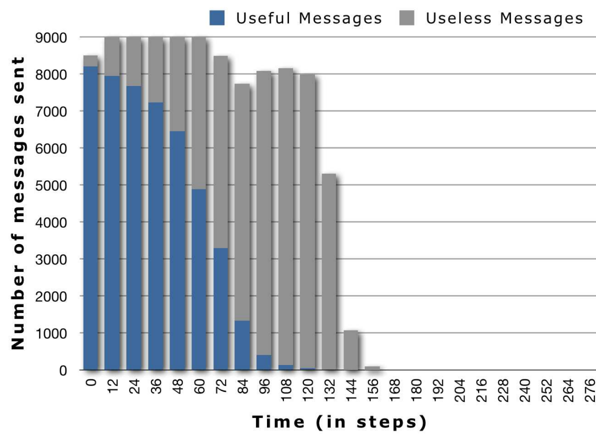


Figure 4.9: Messages generated by the proposed protocol (without the dormant state).

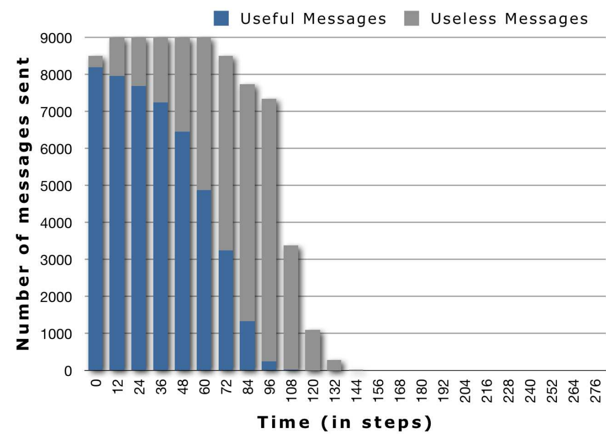


Figure 4.10: Messages generated by the proposed protocol when pessimistic nodes can become dormant.

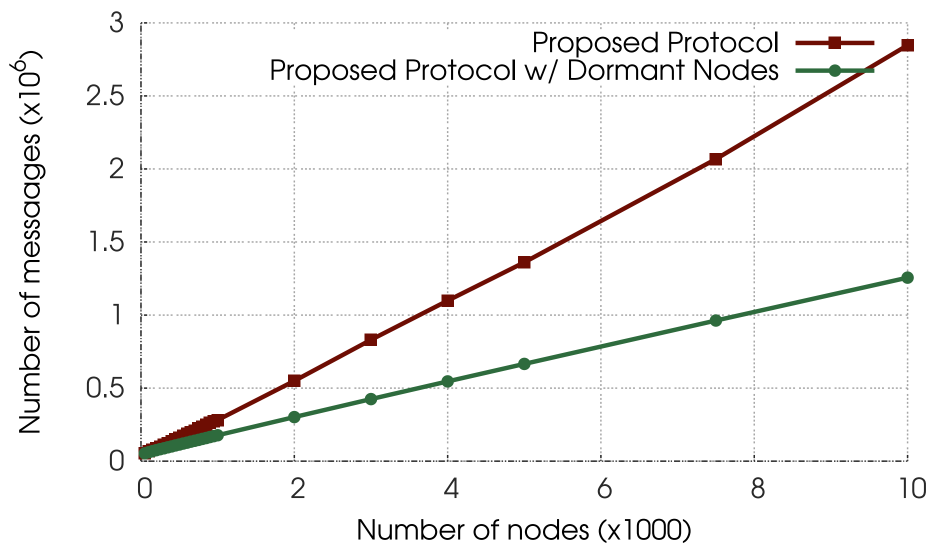


Figure 4.11: Total number of messages sent when varying the number of nodes in the system.

cycle comprises 12 simulation steps, as it is the lowest common multiple of 4 and 6; at most 4 steps are needed for the optimistic sub-protocol to complete, and at most 6 steps for the pessimistic sub-protocol. The messages are classified into two categories: *useful* messages (ones which led to a reaction, in blue) and *useless* messages (those which did not induce a reaction, in grey). When looking at the communication costs of the optimistic sub-protocol (Figure 4.7), one can observe that a high volume of reactions is done in the beginning of the execution with a small percentage of conflicts, and thus a small amount of useless messages. However, as the execution progresses, the percentage of useful messages drops rapidly, while the total number of messages is kept high. Figure 4.8 shows that the pessimistic sub-protocol consumes less messages, with the percentage of useful messages dropping steadily and slowly. At the same time, as there are less and less molecules in the system, the total number of messages slowly grows before peaking at 8000 messages per cycle and then rapidly decreasing towards the end of the execution. When comparing Figure 4.9 to the previous two, we note that the protocol takes over the best properties of both of its sub-protocols. Firstly, it takes over the elevated number of useful messages of the optimistic sub-protocol. After the switch, the pessimistic sub-protocol kicks in, bringing with it a decrease in the total number of messages. Consequently, using a simplistic and lightweight sub-protocol when possible and then falling back on a heavier one reduces network traffic and improves scalability — a decrease of 30% in the number of messages can be observed when compared to the pessimistic sub-protocol alone. In addition, Figure 4.10 reveals that using the policy of dormant nodes further improves the scalability of the protocol, as it significantly reduces the total number of messages towards the end of the execution where there is the highest number of conflicts. Finally, Figure 4.11 shows the number of messages sent when the system's size varies from 50 to up to 10000 nodes and confirms the protocol's scalability : the number of messages linearly grows with the system's size. Moreover, the scalability greatly improves by using dormant nodes — the slope is gentler, rapidly widening the gap between the two curves.

4.7 Experiments with Multiple Rules

Now we are shifting our focus onto the rule-changing mechanism described in Section 4.3. We want to examine its decision-making policy as well as look at the behaviour of the protocol during the execution of programs comprised of multiple rules. There are four experiments in this set, each evaluating one of four multiple-rule programs, the descriptions of which follow.

4.7.1 Multiple-rule Test Programs

Independent-rules Program. A natural extension of the single-rule program, this one contains three rules — $R0$, $R1$, $R2$ — which are *independent*, *i.e.* no two rules consume or produce the same type of molecules. Thus, reactions using these rules can be done fully

concurrently, without any interference, mutual exclusion or synchronisation. The rules consume two molecules of type $T0$, $T1$ and $T2$, respectively. They produce no output.

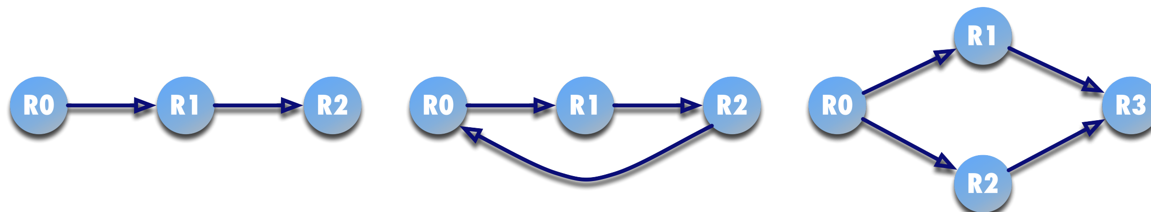


Figure 4.12: Input/output links between rules for the dependent-rules program.

Figure 4.13: Input/output links between rules for the circular program.

Figure 4.14: Input/output links between rules for the workflow program.

Dependent-rules Program. Developing the previous program further we come to the next one. In this new program, the three rules are now *dependent* — the molecules produced by one rule are consumed by another. These input/output links are illustrated in Figure 4.12. Each rule still consumes two molecules of its own type. However, to create the dependencies between them, in this program $R0$ produces two molecules of type $T1$ (used as input by $R1$), while $R1$ produces two molecules of type $T2$ (consumed by $R2$). $R2$ produces no output.

Circular Program. This program exploits the circular-dependency pattern, as shown on Figure 4.13. Its characteristics are the same as those of the dependent-rules program, except that another input/output link has been established between the rules $R2$ and $R0$ in order to create the circular flow diagram: $R2$ now produces a single molecule of type $T0$. Note that the fact that $R2$ produces less molecules (only one) than it consumes (two) ensures inertia will be reached; outputting two would cause an infinite execution loop.

Workflow Program. The last program is somewhat more complex than the others, as it is a small *split/merge workflow* of rules comprising both dependent and independent rules. The links between rules are depicted in Figure 4.14. It consists of four rules: $R0$, $R1$, $R2$ and $R3$. The rule $R0$ consumes two molecules of type $T0$ and produces two molecules: one of type $T1$, the other of type $T2$. These are used as input by $R1$ and $R2$, respectively. These rules can, thus, be run concurrently and independently of each other. $R1$ produces one molecule of type $T3$, while $R2$ produces one of type $T4$. Finally, their outputs are *merged* by the rule $R3$, which consumes one molecule per type — $T3$ and $T4$ — and produces no output.

4.7.2 Execution of the Independent-rules Program

The first program containing multiple rules we examined was the simplest one — independent-rules. Figure 4.15 depicts the flow of its execution: the number of nodes

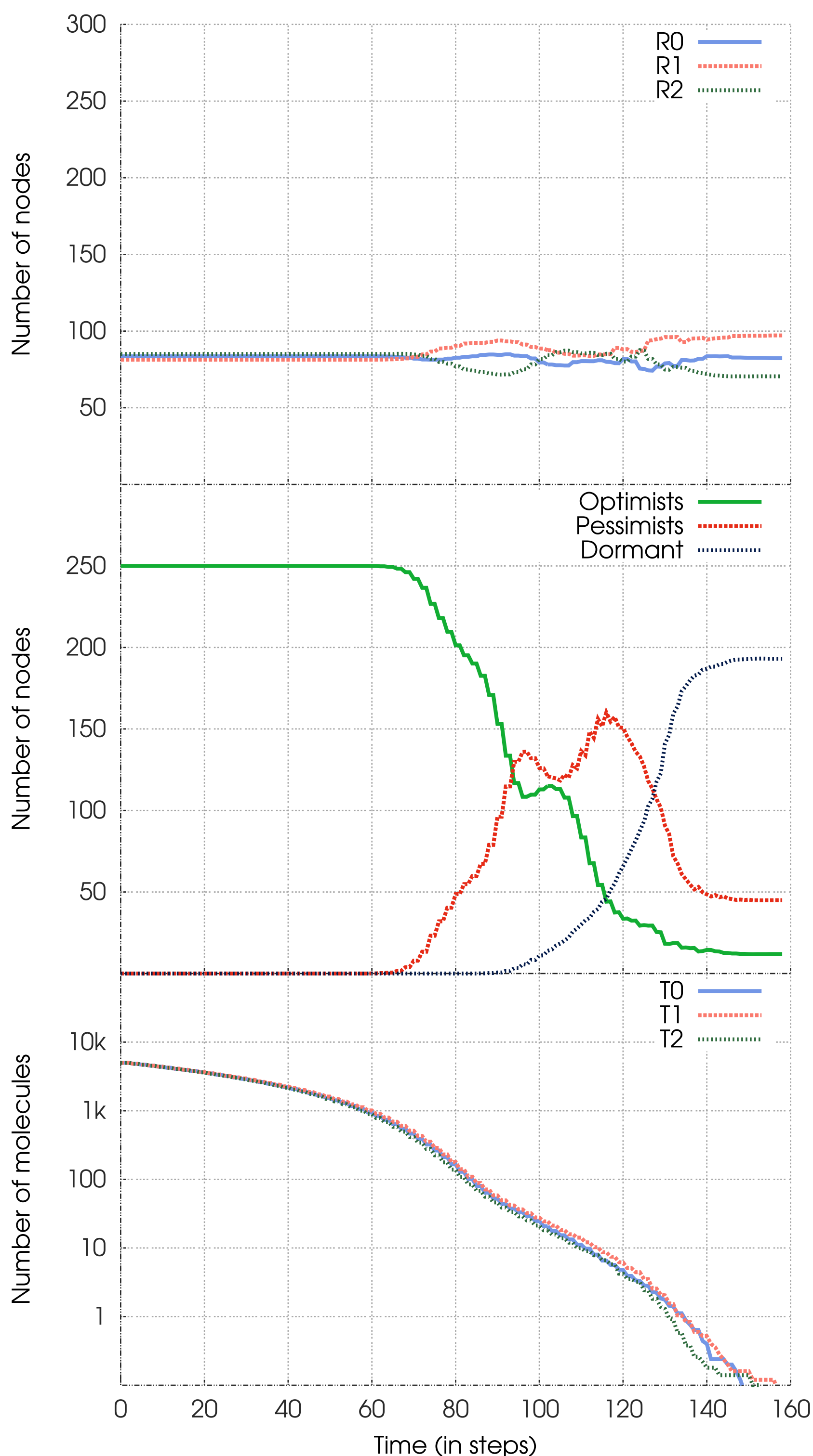


Figure 4.15: Number of nodes executing each rule (top), the number of pessimistic *vs* optimistic nodes (middle) and the number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the independent program.

executing each rule is shown in the top of the figure, in the middle the number of nodes employing each sub-protocol is depicted, while the bottom illustrates, in logarithmic scale, the number of molecules of each type present in the solution. It reveals that an equal number of nodes execute each rule, which is to be expected since all of the rules are executable in parallel and are not in conflict (no two rules share their input molecule types). All the while, all of the nodes employ the optimistic sub-protocol as the concentration of molecules is high enough for nodes to avoid conflicts. As soon the number of optimists starts to decline (around step 75), the nodes start to change rules, causing the fluctuations observed in the upper part of the figure. From that point on we can see a constant decrease in the number of optimists since more and more nodes enter into conflict over molecules, which start to become rarer and rarer. At the same time, there is an increase in the number of pessimistic (and then dormant) nodes, suggesting that most nodes keep employing the pessimistic sub-protocol even after changing rules.

Due to the almost-perfectly equal distribution of nodes over rules they use for reactions we conclude that the rule-changing mechanism correctly decides which rule a node should execute. Moreover, as a result of changing rules, there are nodes employing the optimistic sub-protocol all throughout the execution, in this way speeding up the computation.

4.7.3 Execution of the Dependent-rules Program

Figure 4.16 illustrates the course of the execution of the dependent-rules program. At the beginning of the execution, all of the nodes but $R1$'s and $R2$'s rule keepers are applying the rule $R0$ since there are only molecules of type $T0$ present in the system. Indeed, the discovery protocol is not able to discover molecules of other types, prompting the nodes to change their active rule to $R0$ immediately. Then, as the computation progresses, it becomes harder for nodes to grab $T0$-molecules and they start turning pessimistic. As $T1$-molecules appear, some nodes opt to change their active rule. More specifically, as there are more $T1$-molecules around step 75, most of the nodes choose to execute the rule $R1$, while a minority changes for $R2$ since the rule keeper of $R1$ managed to produce a few $T2$-molecules. Due to this change of rules, all of the nodes become optimistic again. Then, as they successfully perform reactions, the number of $T1$-molecules rapidly drops, inducing another cycle of *mass rule changing*. This time, $R2$ is favoured due to the high concentration of $T2$-molecules. This change prompts nodes to become optimistic again. Because there are some $T1$-molecules left around step 250, half of the nodes change back to $R1$ to complete its execution, causing a sudden drop in the number of optimists and the oscillation between being optimistic and pessimistic. At the same time, the number of dormant nodes increases, meaning that nodes increasingly perceive both rules as pessimistic. This is in accordance with the state of the solution — there are very few molecules left in the system. In spite of the pessimism, towards the end of the execution all of the nodes gradually switch back to $R2$, finishing the execution either as pessimists or dormant nodes.

The conclusion drawn from the experiment is that the local decisions taken by the rule-

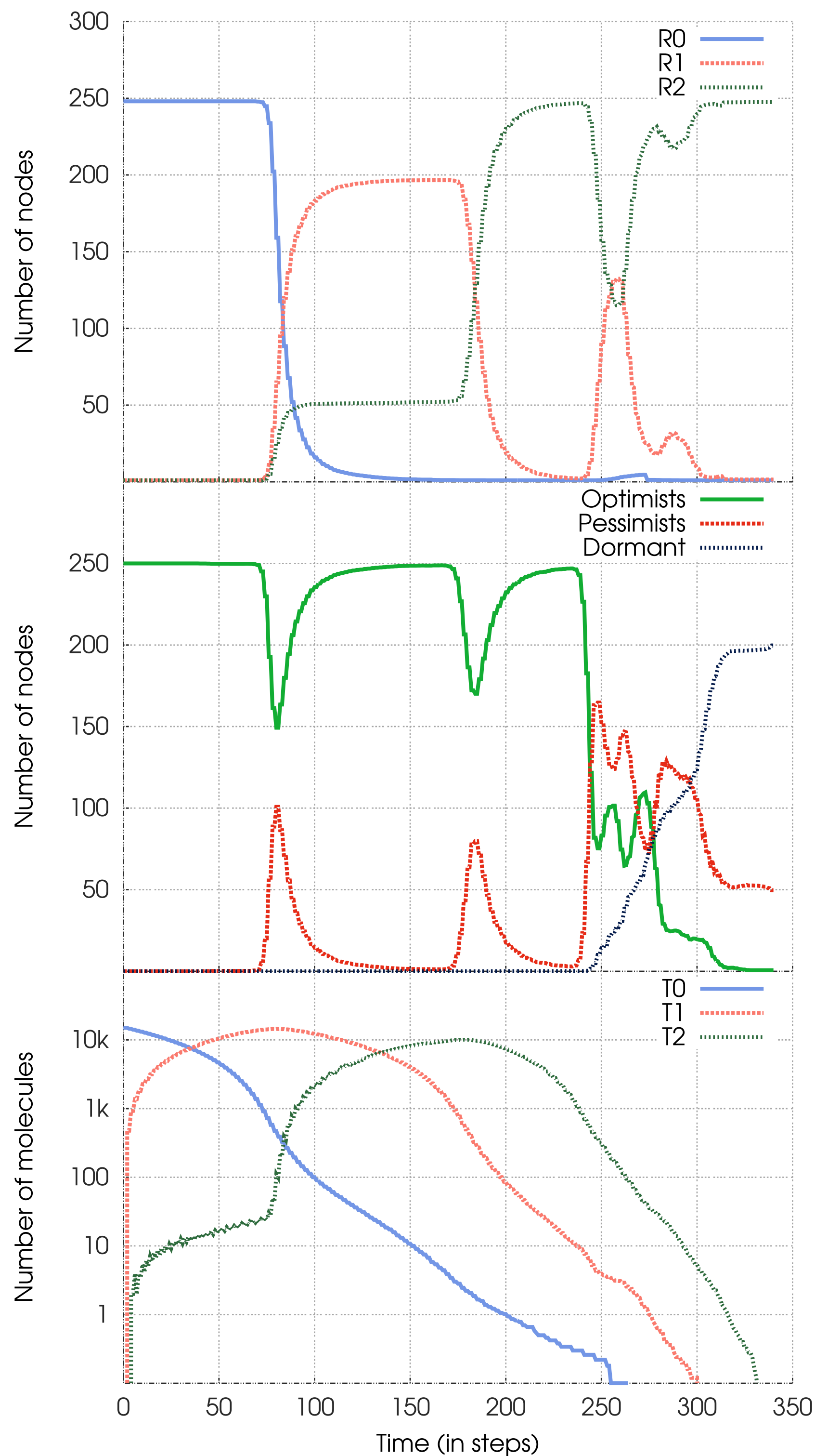


Figure 4.16: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the dependent program.

changing mechanism follow the flow of dependency between rules and are, thus, correctly taken in the case of multiple-rule dependency. In addition, changing rules causes the nodes to become optimistic again, which allows the system to progress faster.

4.7.4 Execution of the Circular Program

Next, we simulated the circular program using two distinct initial configurations, for which the results are shown in Figures 4.17 and 4.18. In the first case the initial solution contained exclusively $T0$-molecules, while in the second all three types of molecules were equally represented.

When examining Figure 4.17, one can see that at the beginning of the execution all of the nodes use the rule $R0$, which is consistent with the fact that there are only $T0$-molecules in the solution. However, the concentration of $T1$-molecules rapidly grows, causing the nodes to pass to the execution of $R1$ once they have become pessimistic. In the same vein, around step 150 they opt for $R2$, after which we can see the $R0$–$R1$–$R2$ execution pattern appear again. As at about step 200 the overall concentration of molecules is rather low, dormant nodes begin to appear, while the number of optimists quickly drops to zero. By the end of the execution, there are only a few molecules left and most of the nodes are thus dormant, while only a small minority completes the few remaining reactions.

The scenery drastically changes when all types of molecules are present in the initial solution, as shown on Figure 4.18. The course of the execution bears a strong resemblance to that of the independent-rules program (Figure 4.15): since all types of molecules are constantly being consumed and produced, the nodes are able to behave as if there were no dependencies between the rules. There are slight differences in the two programs, though. Unlike in the independent-rules program, here one can notice the cyclic change of the rules′ dominance from Figure 4.17 (although on a smaller scale). Furthermore, the number of optimists drops much faster here towards the end of the execution because a decrease in concentration of molecules of one type implies an immediate decrease in concentration of all the others′.

This experiment shows that, while the dependency between rules plays an important role in the course of the execution, so do the data contained in the initial solution when it comes to cyclic dependences between rules. However, both the algorithm and the rule-changing mechanism are able to properly detect the reaction potential of rules in each case, and thus follow the dependency flow brought about by the program.

4.7.5 Execution of the Workflow Program

In the last experiment, we observed the behaviour of the system during the execution of the workflow program. The results, depicted in Figure 4.19, show a substantial similarity to those of the dependent-rules program (Figure 4.16). This behaviour is to be expected, since this program is a variant of the dependent-rules program, whereby instead of one middle rule there are two parallel ones independent of each other.

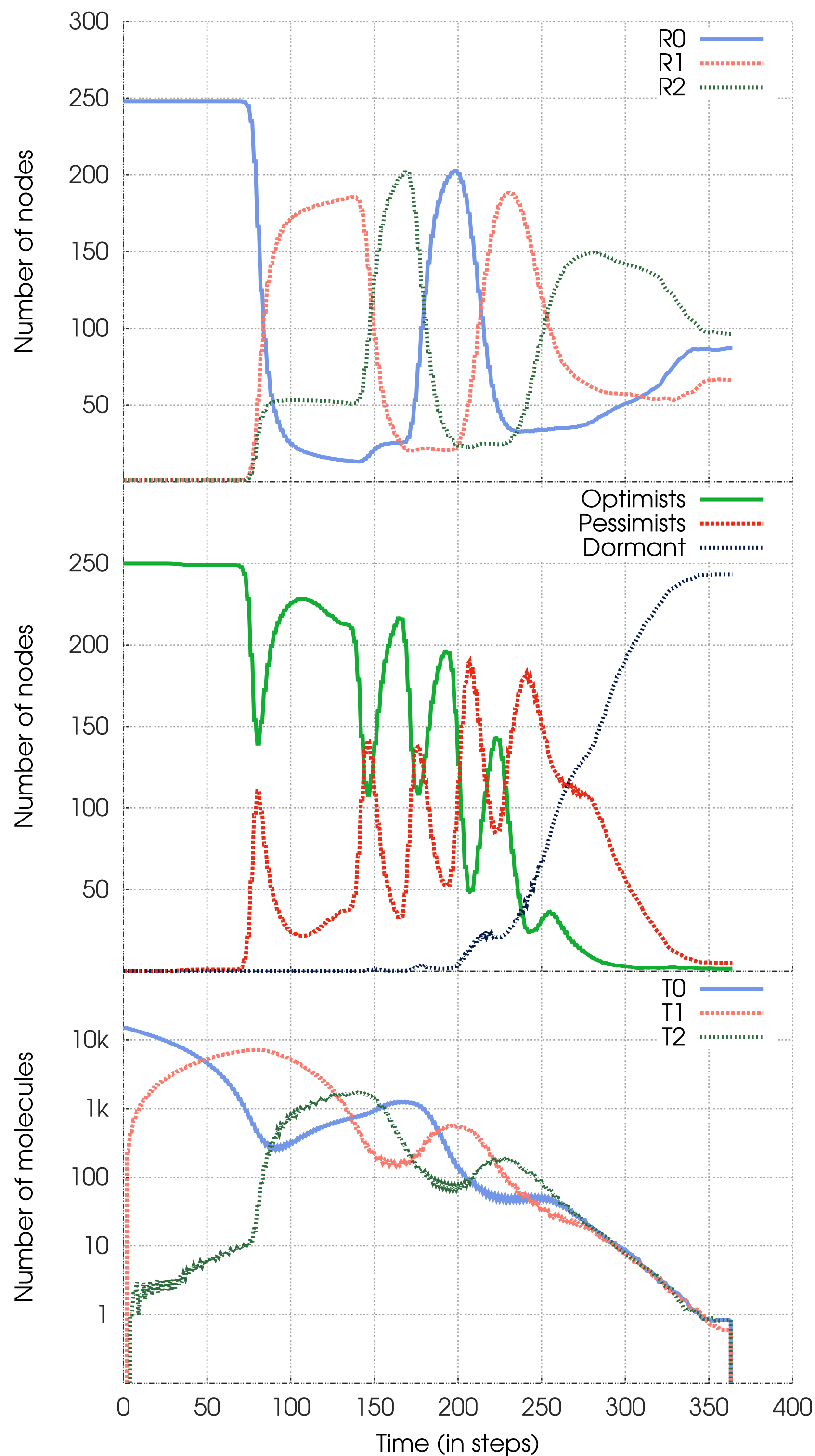


Figure 4.17: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the circular program.

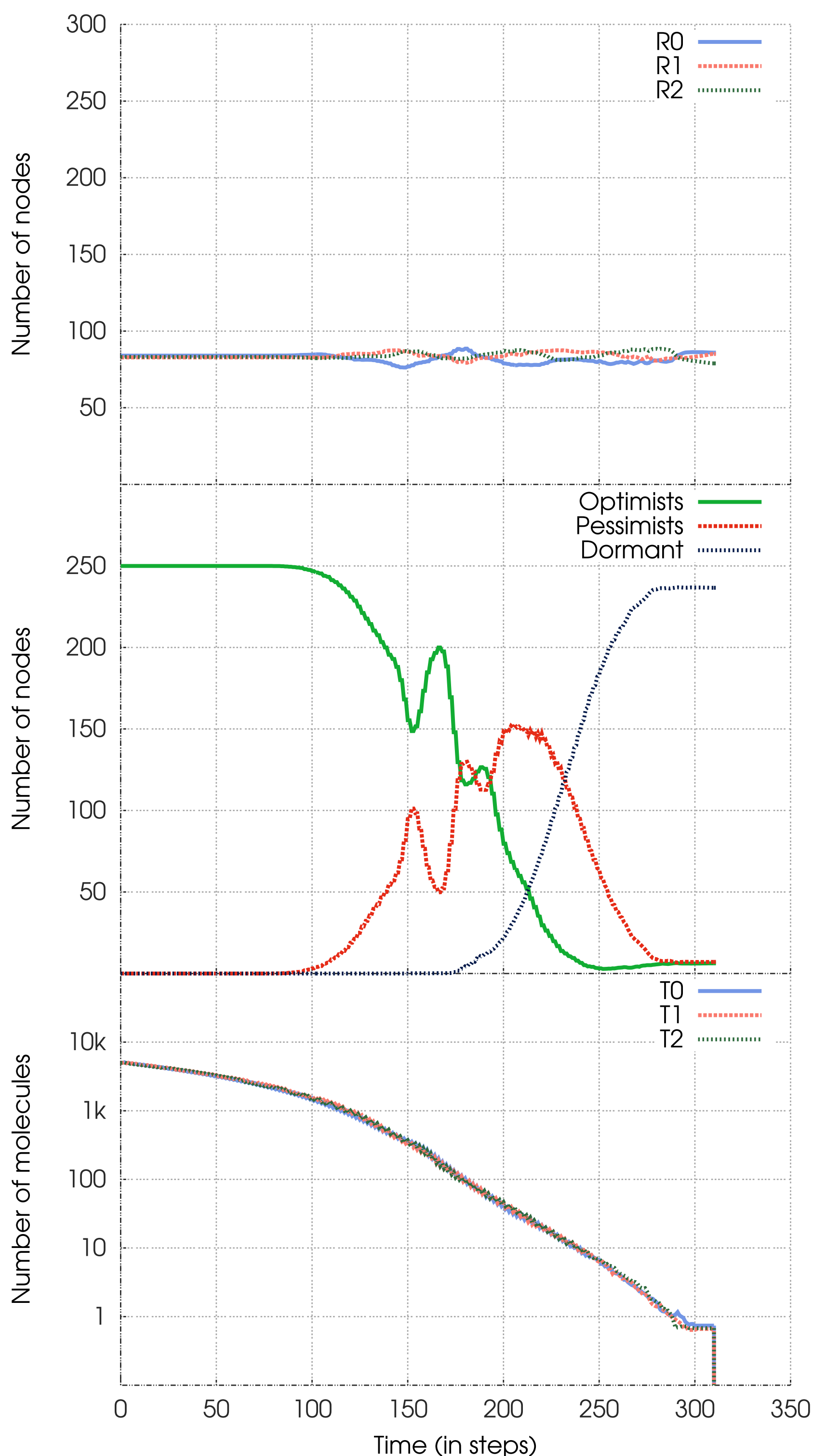


Figure 4.18: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the circular program with different initial conditions.

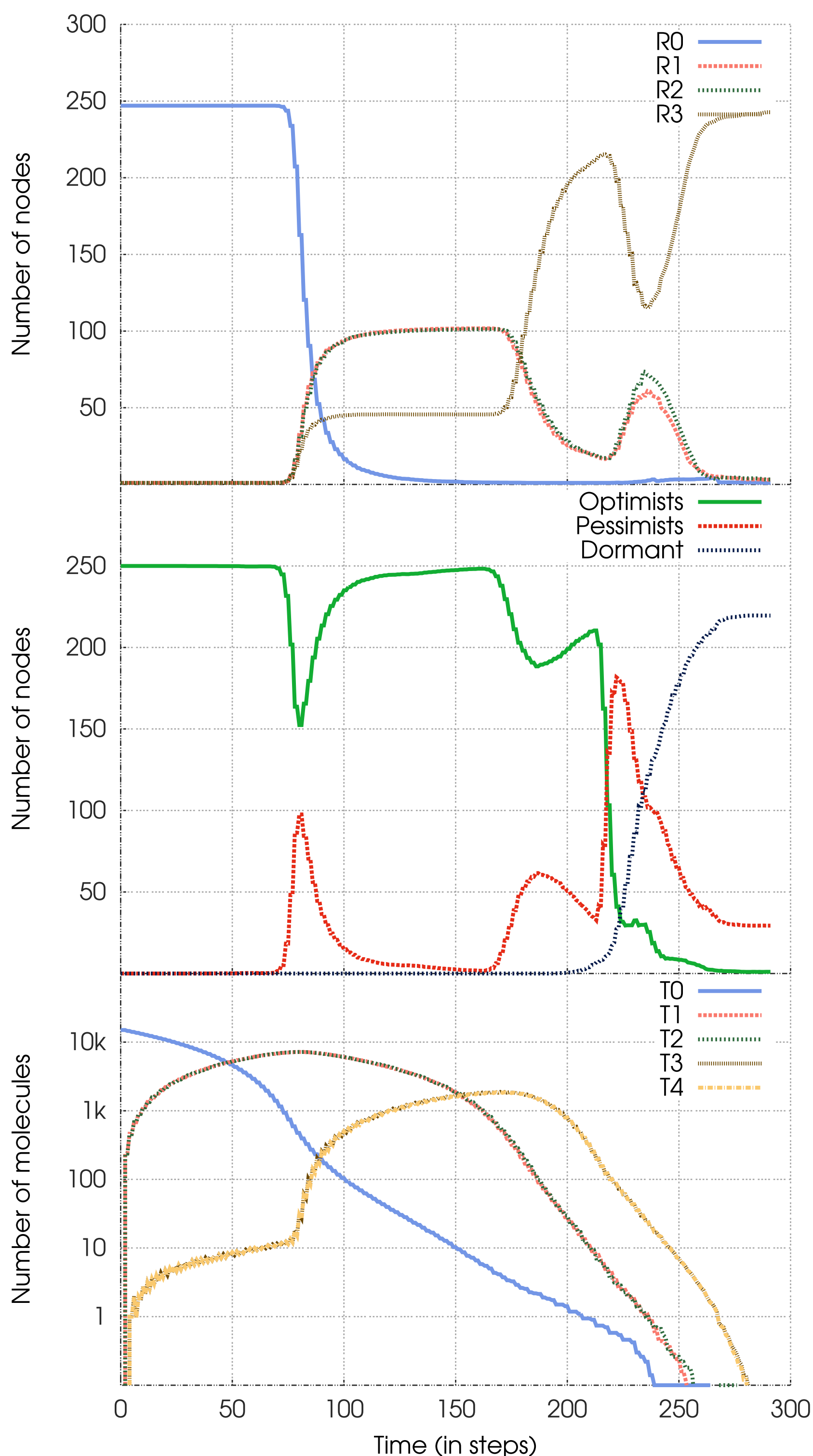


Figure 4.19: Number of nodes executing each rule (top) and the number of pessimistic *vs* optimistic nodes (middle) and the number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the workflow program.

The rule-changing pattern reveals that the nodes first massively execute $R0$ until the concentration of $T0$-molecules drops below those of $T1$- and $T2$-molecules. The nodes then distribute themselves over the rules in such a way as to consume mostly these molecules — around 100 nodes per rule for $R1$ and $R2$. As they produce $T3$- and $T4$-molecules, about 50 nodes start executing the last rule, $R3$. As $T1$- and $T2$-molecules are consumed at a faster pace, nodes start abandoning $R1$ and $R2$ and pick $R3$. All the while, most of the nodes are employing the optimistic sub-protocol. Then, in between steps 200 and 250 about half of the nodes change back to $R1$ and $R2$ in order to consume the remaining $T1$- and $T2$-molecules, respectively, causing a decrease in the number of optimists, due to the globally-low concentration of molecules. At the end of the execution almost all of the nodes use $R3$, with most of them dormant. Indeed, as there are only a few reactions left at that point, it is impossible for most of the nodes to perform reactions in spite the fact that they have correctly picked the rule to execute.

This experiment confirms that the rule-changing mechanism is able to follow more complex dependency patterns. Moreover, we can see that during most of the execution the majority of nodes uses the optimistic sub-protocol, confirming that the protocol is able to adapt itself to the current situation in the system.

4.8 Conclusion

In this chapter, we have described a protocol to capture several molecules atomically in an evolving multiset of objects distributed on top of a large-scale platform. The protocol consists in the association of two sub-protocols intended to face different levels of density of potential reactions in the multiset. By dynamically switching from one sub-protocol to the other, our protocol fully exploits their good properties (the low communication overhead and speed of the optimistic protocol, when the density of reactants is high, and the liveness guarantee of the pessimistic protocol, when this density drops), without suffering from their drawbacks. We also propose a communication-reduction scheme which is activated during the low-density period. Furthermore, we provide a rule-changing mechanism able to guide the nodes' computation when a program with multiple rules is being executed. A formal proof of the protocol's correctness is provided and its different aspects have been experimented with through simulation.

Another interest of this work lies in the fact that it revisits classical problems in distributed systems, but with the specificities of the chemical model in mind. In this way, it tackles the mutual exclusion with the liveness property as a system property while, more traditionally, liveness is a process' property.

Chapter 5

Decentralised Execution Platform

Contents

5.1 Platform Overview **124**
5.1.1 Initialisation 125
5.1.2 Execution 125
5.1.3 Termination 127
5.2 Data Structures and Algorithms **127**
5.2.1 Double DHT Layer 127
5.2.2 Random Meta-Molecule Fetch 129
5.2.3 Search for Candidates 130
5.2.4 Atomic Grab of Molecules 130
5.2.5 Complexity Analysis 131
5.3 Execution of Higher-order Programs **132**
5.3.1 Execution of Rules 133
5.3.2 Correctness of Execution 134
5.3.3 Inertia Detection 136
5.4 Software Prototype **137**
5.4.1 Entities 137
5.4.2 Execution Cycle 138
5.4.3 Optimisations 140
5.5 Evaluation **140**
5.5.1 Test Programs 140

5.5.2 Experimental Results 142
5.6 Conclusion **149**

With the proposition of an efficient and adaptive protocol for the atomic capture of multiple molecules, the previous chapter opened the doors to the realisation of a decentralised runtime for the execution of chemical programs in large-scale environments. This chapter proposes one such execution runtime. More specifically, here we tackle the following problems:

- *Molecule discovery:* molecules are dispatched over the network, meaning suitable reaction candidates have to be found efficiently in spite of the scale of the platform.
- *Decentralised inertia detection:* to secure the termination of a program, we need to ensure to detect the fact that no more reactions are possible, and do so in a decentralised manner.
- *Higher-order capabilities:* the study conducted in Chapter 3 covers only the execution of first-order chemical programs, *i.e.* those in which rules are not considered as being regular molecules. However, in order to leverage the full power of the chemical programming model, the runtime should be able to execute higher-order programs as well, in this way allowing them to be *modified* on the fly during their execution.

This chapter presents the design, development, and experimentation of a generic runtime for higher-order chemical programs. As in Chapter 3, we rely on a DHT to distribute the molecules and rules. We also exploit the DHT's efficient communication protocol as part of a mechanism for discovering molecules. Furthermore, the platform integrates the protocol for the atomic capture of molecules presented in Chapter 4. Finally, the inertia detection is solved through an efficient scheme leveraging a second information retrieval layer built on top of the DHT. For all of the aspects mentioned, the design proposed takes the higher order into account. The viability of the concepts exposed is showed through a complexity analysis. The platform is then put into practice through a software prototype and its experimental validation.

5.1 Platform Overview

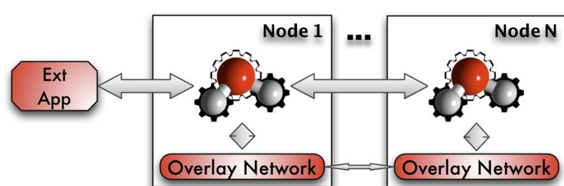


Figure 5.1: The platform.

The external view of the platform (Figure 5.1) resembles that of the hierarchical one described in Chapter 3: the external application hands the chemical program to a node participating in the platform — the *source node* — and then, using the DHT communication protocol, the nodes of the platform execute the program. Naturally, the platform can execute multiple programs concurrently. For the sake of parallelism and load balancing, each of them can have a different *entry point* in the system.

5.1.1 Initialisation

During the initialisation process, molecules — the data — are dispersed in the network and *meta-molecules* — the meta-data — are created. The details of this step now follow.

5.1.1.1 Data Distribution

After receiving the data, the source node scatters the data molecules across the system uniformly at random according to the DHT's hash function and broadcasts the program's rules. By tracing the molecules' paths, a tree, rooted at the source node, is created. Once all of the molecules have been routed, the source node uses this *multicast tree* to diffuse the rules contained in the program. In this way, each node holds a subset of the program's data molecules with high probability if the number of molecules is high enough, and all of the rules, enabling a high level of parallelism and concurrency in performing reactions.

5.1.1.2 Meta-data Creation

Since molecules are spread throughout the system, nodes must be able to find suitable candidates for reactions. In order to efficiently detect inertia, and consequently increase the platform's scalability, with regard to both the number of nodes as well as the number of molecules, we use a second DHT layer. It contains *meta-molecules*, which are placed around the key space in an order-preserving manner, allowing participants to search for the existence of a particular molecule, or a set thereof, during the execution phase.

Each node produces meta-molecules based on its local molecules and routes them according to their order-preserving identifier. Each molecule is associated a state in its meta-molecule. Initially a meta-molecule's state is set to *free*, indicating that nodes can freely take the molecule it describes and combine it with other molecules to perform reactions. At a later stage, during execution, a meta-molecule's state may be set to *inert*, which denotes that a suitable combination for its molecule has not been found thus far.

After it dispatches all of the meta-molecules and receives the rules from the source node, a node is ready to enter its execution phase.

5.1.2 Execution

The distributed platform presented here adopts a reactant searching scheme , in which the system is explored for molecules with specific properties matching a rule's pattern and condition, such as *an integer greater than 3*, allowing it to detect inertia more efficiently.

The main execution loop, executed by every node, is described in Algorithm 5.1. For the sake of clarity, the algorithm is presented in a simplified form, in which only one rule involving a pair of molecules is considered. Nevertheless, it is easily expandable to multiple rules and multiple molecules per rule. This section gives an overview of the execution, the details of the concrete routines used are provided throughout Section 5.2, while Section 5.3 focuses on the implementation of the higher order. The execution consists of

three steps: *(i)* getting a random meta-molecule and testing inertia (lines 2—4), *(ii)* finding a candidate molecule it can react with (lines 5—9) and *(iii)* atomically grabbing the corresponding molecules and performing the reaction (lines 10—14).

During the first step, a node tries to obtain a random meta-molecule, the state of which is *free*. `random_mol` guarantees a *free* meta-molecule will be returned, in case one exists. If, on the other hand, no meta-molecule can be found, it means that, previously, another node could not find any candidate to react with the currently present molecules, implying their states were set to *inert*. This signals to the requesting node that inertia has been reached. It then stops executing the main loop (line 4).

Obtaining a free meta-molecule triggers the second execution step (lines 5—9). The node now asks the system to find it a suitable meta-molecule by supplying the meta-molecule found in step one and the rule which needs to be applied on the molecules to the `find_candidate` routine (line 5). This routine systematically searches for a meta-molecule matching the provided rule's pattern and reaction condition. It is important to note that, in this routine, both *free* and *inert* molecules are checked. If no suitable candidate has been found, then the state of the first meta-molecule obtained is changed to *inert* and stored back in the second DHT layer.

Algorithm 5.1: Main execution loop.

```
1  while not inert do
2      meta_mol1 = random_mol(state = free);
3      if meta_mol1 = null then
4          break;
5      meta_mol2 = find_candidate(meta_mol1, rule);
6      if meta_mol2 = null then
7          meta_mol1.state = inert;
8          store(meta_mol1);
9          continue;
10     if grab_molecules(meta_mol1,meta_mol2) then
11         execute_reaction(rule, mol1, mol2);
12         store(new_mol1, new_mol2);
13         store_ack(meta_new_mol1, meta_new_mol2);
14         remove(meta_mol1, meta_mol2);
```

Step three (lines 10—14) concludes an execution loop iteration. The node tries to grab atomically the molecules described by the previously obtained meta-molecules. If the `grab_molecules` routine succeeds, it ensures no other node will obtain these molecules, making it possible to trigger the actual execution of the reaction, after which the meta-molecules describing the newly created molecules are produced. The new molecules are then sent to their respective nodes based on their hash identifiers. It is important to note that, to store the meta-molecules, the `store_ack` procedure is used, which blocks the execution until the node receives the confirmation of their arrival at their respective destinations. Only then the old meta-molecules are removed in order to ensure inertia is not falsely detected by another node.

Algorithm 5.1 outlines the steps only for one rule and two molecules. However, in its full form, the rule-changing mechanism described in Section 4.3 is used to select which of the rules received from the source node is going to be used. Additionally, if the active rule takes more than two arguments, the instructions of the second step (lines 5—9) are iter-

atively repeated until the right amount of candidate meta-molecules has been obtained. In case suitable candidates cannot be found, the rules are sorted in a descending order based on their perceived reaction potential and step two is repeated for all the rules in the list before marking the meta-molecule found in step one as *inert*.

5.1.3 Termination

Inertia has been detected once `random_mol` (Algorithm 5.1, line 2) can no longer find a *free* meta-molecule in the system. This marks the end of execution and the beginning of the termination phase. On inertia detection, each node sends its molecules back up the multicast tree, after which the source node transfers the now inert solution to the external application.

Theorem 5. *If a node detects inertia, global inertia has been reached.*

Proof. The global property of inertia arises from the distribution of molecules and meta-molecules: if one node cannot find *free* meta-molecules, and consequently molecules, to work with, others will not be able to find them either. However, we need to prove that a node cannot declare the state of inertia while another node performs a reaction, *i.e.* that there are no *false positives* in inertia detection. The proof is visible in Algorithm 5.1. Assume a node n is executing the lines 10—14 and has been able to obtain both molecules needed for a reaction, while another node n' is beginning a new loop iteration. Further, assume either `meta_mol1` or `meta_mol2` or both are the only *free* meta-molecules present in the system. Then, node n will be executing the reaction and storing its products. Meanwhile, node n' will obtain either `meta_mol1` or `meta_mol2` from `random_mol` and will be able to find a candidate for it (the other meta-molecule) although their molecules do not exist any more. This happens due to node n's call to the blocking `store_ack` routine before removing the old meta-molecules. Thus, while at least one node is executing a reaction, others will not be able to declare the state of inertia. □

5.2 Data Structures and Algorithms

This section presents in detail the data structures used and algorithms employed by the platform in order for the nodes to be able to execute a chemical program as outlined in Section 5.1. The underlying data structures are first described, and are followed by an in-depth explanation of each of the central routines in Algorithm 5.1.

5.2.1 Double DHT Layer

Due to the versatility of molecules, their sizes can vary considerably, potentially provoking network overhead if moved frequently. As introduced in Section 5.1, in order to reduce superfluous network traffic, on top of the Pastry ring (the *uniform layer*) we place a second one containing *meta-molecules* positioned in an order-preserving manner (the

order-preserving layer). The logical placement of the two layers is visible on Figure 5.2. Since both layers share the same key space, each node is in charge of both molecules and meta-molecules residing within its responsibility area.

5.2.1.1 Uniform Layer

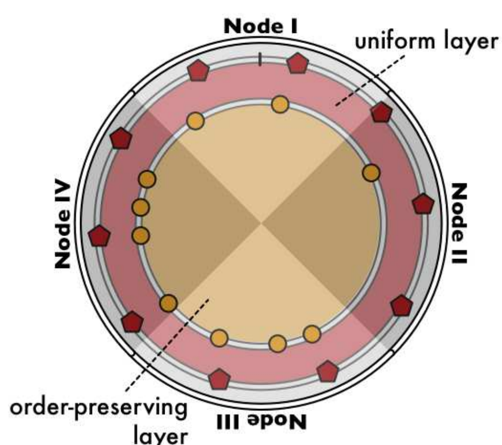


Figure 5.2: Double layer: the key space of the uniform layer coincides with that of the order-preserving layer. The alternating grey and white regions designate responsibility areas of different nodes.

The first of the two layers uses a cryptographic hash function, such as SHA-1, to spread the original molecules uniformly around the system. The source node initially scatters the data molecules across the system according to their hash values. Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops, where n denotes the number of nodes. In the course of the routing process, the path of each molecule is traced by intermediary nodes, in this way creating a *multicast tree* rooted at the source node. As soon as all of the molecules are scattered, the source node uses this tree to diffuse the rules contained in the solution to every node. Note that, due to the uniform dispersion of molecules (w.h.p.), every node will receive the rules and will be able to execute them later on. This multicast tree is conserved and used later during the termination phase to deliver all the remaining molecules residing in the uniform layer to the source node, which hands it over to the external application.

The only other time this layer is utilised in the execution phase is during step three: once a combination of molecules able to react with a rule has been found, the molecules are moved from their original nodes to the one which will execute the reaction, *i.e.* the molecules are effectively removed from the layer. At the end of the reaction, the newly created molecules are hashed with the uniform hash function and also put into this layer.

5.2.1.2 Order-Preserving Layer

The placement of the second DHT layer coincides with that of the first one: both use the same key space and nodes keep their identifiers as well as their routing tables. Still, it is used for storing meta-molecules — objects which describe the actual molecules present in the system. Each of them carries four pieces of information:

- the hash identifier of the original molecule in the first layer;
- its type;
- its cardinal position in the molecule's type's total order; and

- its state — *free* or *inert*.

In order to quickly and efficiently locate *free* meta-molecules, the key space is split in two parts: one containing only *free* meta-molecules, within the range $[0, \frac{ks}{2} - 1]$, and the other consisting of only *inert* meta-molecules, within the range $[\frac{ks}{2}, ks - 1]$, where ks is the size of the key space. Both halves of the key space are organised in the same way: the position of a meta-molecule is based on the total ordering of values of a specific molecule type.

A meta-molecule's identifier in the second layer is calculated as:

$$id = \begin{cases} \frac{cv}{|\mathcal{V}|} * \frac{ks}{2} - 1 & \text{, for } state = free \\ \frac{ks}{2} * (1 + \frac{cv}{|\mathcal{V}|}) - 1 & \text{, for } state = inert \end{cases}$$

where cv is the molecule's value's cardinal position in the total order and $\mathcal{V}$ is the set of all possible values the molecule can have. As a consequence, when a meta-molecule's state changes, its identifier is recalculated, relocating it to the *other* half of the key space, as illustrated by Figure 5.3. Note that multiple types of molecules reside in the key space — it is not divided amongst them. Thus, different meta-molecules may have the same identifier, but they can be distinguished by their molecule types.

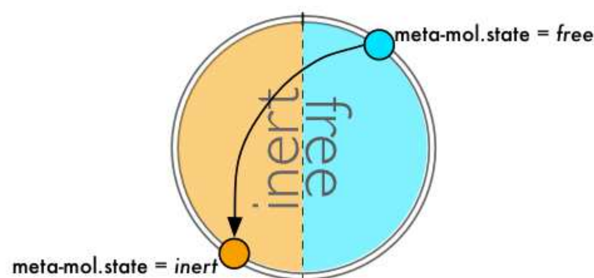


Figure 5.3: Order-preserving layer: as a meta-molecule's state changes, it repositions itself in the second layer.

Organising the layer in this manner lets nodes search exactly for the reactants they need for completing reactions. They do so by performing range queries [22, 111] on the second layer.

5.2.2 Random Meta-Molecule Fetch

The starting point of the execution phase is obtaining a meta-molecule by means of the `random_mol` routine (Algorithm 5.1, line 2). In order to avoid multiple nodes attempting to fetch the same meta-molecule, each node tries fetching a randomly chosen meta-molecule, which in turn improves load-balancing and reduces the risk of bottlenecks. To that end, `random_mol` chooses a random identifier within the range $[0, \frac{ks}{2} - 1]$ since the node is looking for a *free* meta-molecule. The requesting node sends a *METAMOL_REQ* message request, comprising the chosen identifier, the node's identifier and the range it wishes to search — the whole half key space —, to the corresponding node. The receiver of the request checks its local meta-molecules and returns it the one with the closest identifier. The method is exemplified in Figure 5.4.

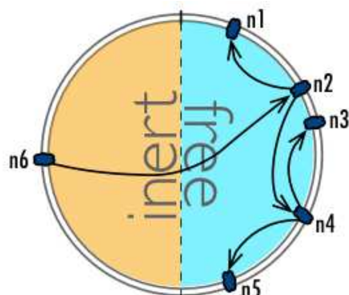


Figure 5.4: Meta-molecule fetch example: $n6$ asks $n2$ for a random meta-molecule which then forwards the request to $n1$ and $n4$. The process is repeated until a *free* meta-molecule is found.

In case the receiver of the request does not hold any *free* meta-molecules, it splits the search range into two parts: $[range_beginning, min_id - 1]$ and $[max_id + 1, range_end]$, where $[min_id, max_id]$ denotes receiver's responsibility area. It then generates one random identifier for each range and sends two *METAMOL_REQ* message requests. This process continues until a meta-molecule has been found or until the whole half key space has been searched with no result. In the latter case, a *NO_METAMOLS* message is sent to the original requester, after which the termination phase is triggered on both the original requester and the original receiver of the request.

5.2.3 Search for Candidates

With the order-preserving layer in place, finding a suitable candidate for a reaction becomes a matter of successfully interpreting the rule to be triggered and associating it the previously retrieved random meta-molecule. The rule, of the form **replace** *P* **by** *M* **if** *V*, is analysed and the type of the candidate is first extracted from the pattern *P*. Next, the random meta-molecule is introduced in the reaction condition *V*; the cardinal position of the value replaces its symbol. Such a modified rule is then tested to establish the values which would satisfy the reaction condition. These values are aggregated into ranges of values and enclosed each in a *METAMOL_REQ* message along with the desired type.

The process of finding a matching meta-molecule in the system conforms to the previously described method of locating random meta-molecules, only this time the ranges cover the whole key space. Thus, at least two *METAMOL_REQ* messages are sent: one targeting *free* meta-molecules and the other *inert* ones. If neither of the requests returns a meta-molecule, the requester sets the initial random meta-molecule's state to *inert* and stores it in the second half of the key space.

Note that, in case a range is fragmented, each part is enclosed in a different *METAMOL_REQ* message. Thus, the requester may receive more than one meta-molecule suitable for a reaction. The node then chooses randomly one of the arrived meta-molecules, discarding the rest.

5.2.4 Atomic Grab of Molecules

Finally, the last step of the execution phase consists in obtaining the actual molecules discovered and consequently performing the reaction. Given the fact that a molecule can be consumed only once, *i.e.* it can be used in at most one reaction during its lifetime, it is imperative that nodes grab all of the molecules in an atomic fashion, as several nodes can

discover the same molecules concurrently. For this task, we rely on the protocol described in Chapter 4. It is implemented in the `grab_molecules` routine (Algorithm 5.1, line 10) and its outcome determines whether a reaction will take place or not.

Once the reaction has been performed, new meta-molecules are created based on the newly produced molecules. The new molecules are then hashed and stored in the first layer, while the new meta-molecules are stored in the second layer using the blocking `store_ack` routine. In case the reaction did not take place, the node simply discards the meta-molecules and molecules it has obtained and starts a new execution cycle.

5.2.5 Complexity Analysis

We now provide a complexity analysis of the time and network costs of running a chemical program on top of the proposed platform. Due to the versatility of the chemical paradigm and the volatility of the execution (asynchronism, conflicts during molecule captures), a precise analysis of the general case is not feasible. For this reason, in the following, we restrict our analysis to the execution of a chemical program containing only reaction rules reducing the number of molecules. While this may appear as a limitation, note that this behaviour mimics most data-processing applications and services, where multiple input values are processed to produce one output value. We assume the presence of a single rule with r arguments acting upon m molecules in a system comprising n nodes uniformly arranged across the key space, with $m \gg n \gg r$. The rule produces a single molecule as its output.

5.2.5.1 Execution Time Analysis

The costs of the initial dissemination and the termination phases can be easily calculated. The former depends on the number of molecules to be disseminated, thus running for $O(m)$ time. The latter is conditioned by the number of nodes having to return their molecules, putting its cost to $O(n)$.

The number of loops of Algorithm 5.1 each node will do is proportionate to $\frac{m}{r}$. However, because $m \gg n$, at the beginning of the execution every node will be able to capture an exclusive set of r molecules. As the computation progresses, more and more conflicts will arise, linearly decreasing the number of nodes able to complete a reaction as the number of available molecules linearly decreases with each reaction, until there are only r molecules left for which all of the nodes will compete. Consequently, on average $\frac{n}{2}$ nodes perform a reaction in each iteration, while the rest aborts theirs. Thus, each node will loop through Algorithm 5.1 $\omega = \frac{2m}{rn}$ times, consuming $O(\frac{m}{n})$ units of time.

5.2.5.2 Network Traffic Analysis

The initial dissemination generates $2m \log n$ messages — for m molecules and their meta-molecules and the dissemination of rules —, *i.e.* its network cost is $O(m \log n)$. The termination phase needs $2n$ messages, putting its cost to $O(n)$.

In the worst-case scenario, which happens towards the end of the execution phase when there are only a few *free* meta-molecules left, `random_mol` and `find_candidate` generate at most $2\log n + n + \log n$ messages each. $2\log n$ messages are needed for sending the request and receiving the response; n messages are spent to search the entire key space and $\log n$ messages are used for returning a molecule to the first node in the key space contacted by a requester.

In the final step of the execution loop iteration, we consider only the heavier of the two sub-protocols for grabbing molecules, since it generates more network traffic. The pessimistic sub-protocol needs $6r\log n$ messages to complete its three phases ($2\log n$ messages per phase per molecule). As explained above, $\frac{n}{2}$ nodes perform reactions, completing all the three phases of the sub-protocol. This situation produces $10r\frac{n}{2}\log n$ messages in total. The nodes which have grabbed the r molecules generate further $(3+r)\frac{n}{2}\log n$ messages. $\frac{n}{2}\log n$ messages are used to disseminate the new molecules, $2\frac{n}{2}\log n$ messages are needed for storing the new meta-molecules and their acknowledge receipts, while $r\frac{n}{2}\log n$ messages are spent to remove the old meta-molecules. Since the number of such cycles needed to reach inertia equals to ω, the cost of the execution phase is $O(mn\log n)$.

Understandably, the execution phase generates the highest number of messages when compared to the other phases. Consequently, we approximate the cost of a complete program execution in terms of network cost to be $O(mn\log n)$. Bearing in mind the factorial complexity of the traditional centralised method, this analysis shows the plausibility of executing chemical programs on the proposed platform with a network traffic overhead proportional to the program's size and to the number of nodes, establishing the benefits of such a platform.

5.2.5.3 Concluding Remarks

Seeing that the platform may be used for coordination purposes as well, the number of molecules in the system might be considerably lower, *i.e.* $m \approx n$. In such a scenario, the number of loops done by each node is $\omega \approx \frac{n}{r}$, *i.e.* the execution time is $O(n)$. Furthermore, only one node is able to complete a reaction in one loop pass and thus generating $(4n^2+3n)\log n$ messages, elevating the network cost to $O(n^2\log n)$. However, because in such a regime m is rather low, the execution lasts for a reasonably small amount of time, presenting only short-term spikes in network load.

5.3 Execution of Higher-order Programs

Higher-order programs contain rules which manipulate rules as if they were ordinary molecules — *rule molecules* can be consumed as well as produced in reactions. Still, there exist differences between rule molecules and ordinary ones. Namely, the former are *n-shot* while the latter are *one-shot*. Once a regular molecule is captured and consumed, it disappears from the multiset. On the other hand, when a rule is being executed, its

molecule only has to be present in the solution but it is not consumed in the process, so it can be applied in parallel by multiple nodes.

This section details the extensions incorporated into the decentralised runtime presented in Section 5.1 to allow it to execute higher-order programs. Namely, we address here two crucial issues:

1. the discovery of present rules and their execution; and
2. handling potential inconsistencies arising when rule molecules are removed from the multiset.

5.3.1 Execution of Rules

As rules can be manipulated (and thus consumed) by other rules, they can appear in and/or disappear from the multiset. Hence, even though each node knows all of the rules the program *may* contain — they are defined statically by the programmer —, they cannot be simply executed due to the volatility of the rules' molecules — they might or might not be present in the solution at the time of execution. Therefore, a node has to check whether the rule it is about to execute exists or not.

5.3.1.1 Tracking Rules' Existence

When a node receives the initial program at the beginning of the computation, it labels every rule defined in it as *non-existent* since it does not know which of their molecules actually exist in the solution. Each rule in the *non-existent* list is assigned a recheck time — a timestamp stating when should a node recheck whether the rule's molecule has appeared in the solution. At the beginning of each execution cycle, before searching for a random meta-molecule, a node goes through this list and singles out the rules which have to be rechecked for existence. To do so, for each rule, the node sends an *ALIVE* message to the holder of the rule's molecule. If the response is negative, a new recheck time is assigned to the rule, leaving it in the *non-existent* list. In case the holder replies with a positive message, the rule is transferred to the *existent* list, which keeps track of rules the molecules of which are present in the solution. The execution cycle then resumes by selecting a random meta-molecule.

Furthermore, if the node is a holder of *inert* meta-molecules, it checks whether there are local meta-molecules which could satisfy the pattern of the rules recently added to the *existent* list. Those meta-molecules are then redeclared as *free*, *i.e.* their state is changed and they are stored in the *free* half of the order-preserving layer, in this way allowing all of the reactions with the newly-existent rule to be performed.

5.3.1.2 Rule Existence Confirmation

The rule to execute is chosen from the list of existent rules according to the policy specified in Section 4.3. If candidate meta-molecules can be found for it, the matching molecules

are grabbed. Between the time the node last checked for a rule's molecule's existence and the time the molecules have been captured, the rule's molecule might have been consumed by another rule on another node. Thus, before actually performing the reaction the node inquires the holder of the rule's molecule's presence in the solution. The reaction is carried out if the response is positive. Otherwise, the rule is placed in the *non-existent* list and assigned a new recheck time, while the captured molecules are returned to their respective holders.

5.3.1.3 Treatment of Rule Molecules

While they are still molecules, rule molecules are handled a bit differently than regular, data ones, since they represent the functional part of the program. The time diagram of the execution of a higher-order rule, one which consumes a rule molecule, is shown in Figure 5.5. $R1$ is an *ordinary* rule consuming *ordinary* molecules, and $R2$ is a higher-order rule, as it consumes the molecule of $R1$ as part of its reaction. In Figure 5.5, $c(R1)$ denotes the node about to execute the rule $R2$, $h(R1)$ represents the node holding $R1$'s molecule, while $h(R2)$ is the node holding the molecule of $R2$. First, $c(R1)$ captures the molecule of $R1$ and then checks for the existence of $R2$ by sending an *ALIVE R2* message to $h(R2)$. After it has received the confirmation, $c(R1)$ is sure that a reaction is going to be performed. Thus, it informs the holder of $R1$ that a reaction is about to be performed with a *DONE* message, and then carries out the reaction.

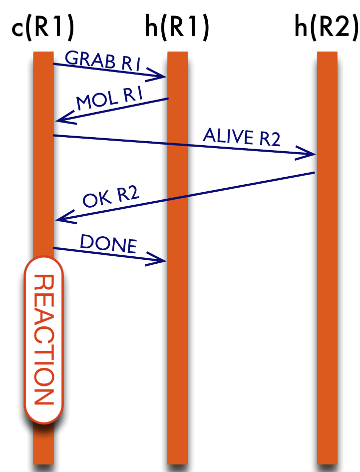


Figure 5.5: Time diagram for higher-order rules.

This last step is done only when a rule molecule is consumed in a reaction. It is needed in order to confirm to the holder that the rule molecule does not exist any more. In between the time it gives the rule molecule to a node and the time it receives the *DONE* message, a holder continues to reply positively to *ALIVE* requests, since it cannot be sure whether the reaction consuming the rule molecule it has given away has been performed or not; the mere fact of capturing a molecule does not ensure a reaction will take place. While this step may introduce inconsistencies during execution, it is needed in order to guarantee the proper detection of inertia. We show below why these inconsistencies do not disrupt the computation's correctness and how does the platform ensure the correct detection of inertia.

5.3.2 Correctness of Execution

The chemical programming model enforces mutual exclusion of molecules — a molecule cannot be used in more than one reaction. The same, naturally, applies to rule molecules — once it has been consumed in a reaction, the rule cannot be used for execution. Because the execution platform proposed here is an asynchronous and a decentralised one, there may be situations in which exactly the opposite happens,

i.e. where a rule is used to perform a reaction *after* its molecule has been consumed. By using the characteristics of the communication pattern in the runtime and the properties of the chemical programming model, we show that the execution provided by the platform complies with the execution model of the chemical paradigm in spite of the inconsistencies which might arise during execution.

Theorem 6. *The execution provided by the decentralised runtime conforms to the specifications and requirements of the chemical programming model.*

Proof. Consider a network with three nodes — N_1, N_2 and N_3 — executing a program containing two rules — R_1 and R_2. Let us assume R_1 consumes an ordinary molecule of an arbitrary type and a molecule of type R_2, *i.e.* R_2's molecule if it is present in the multiset, while R_2 consumes two ordinary molecules:

$$\textbf{let } R_1 = \textbf{replace } x :: gen, y :: R_2 \textbf{ by } x$$
$$\textbf{let } R_2 = \textbf{replace } x :: gen, y :: gen \textbf{ by } x$$

where *gen* denotes a generic type excluding rule molecule types. Let us assume, furthermore, that N_1 wants to carry out a reaction using R_1, N_2 holds R_2's molecule and N_3 is about to perform a reaction using R_2. Both N_1 and N_3 will go through the same process: first, they are going to capture the molecules needed for their respective reactions and then inquire about the existence of the molecule of the rule each of them is about to execute. Thus, both nodes will, at some point, contact N_2; N_1 will try to obtain R_2's molecule to consume it, while N_3 will check for its existence. This leads to the following possible scenarios:

1. N_3's check request reaches N_2 before N_1's capture request. This is a simple case: N_2 simply replies positively to N_3 and then hands the molecule to N_1. Consequently, both nodes are able to carry out their reactions.

2. N_3's check request reaches N_2 after N_1's capture request, but before N_1 reports whether it has performed the reaction or not to N_2. As noted earlier, after it has given the molecule to N_1, N_2 is still going to reply positively to N_3's check request, and it will, thus, carry out its reaction. There are two possible sub-scenarios:

   - N_1 has aborted its reaction. It gives R_2's molecule back to N_2, in this way making it available again. Hence, no inconsistencies have arisen as a result of N_2's positive reply to N_3.
   - N_1 has been able to perform its reaction. Thus, an inconsistency might arise if, in the sequential order of events, N_1 carried out its reaction before N_3 did, since that would mean N_3 executed a reaction of a non-existent rule. We will address this case in the next paragraph.

3. N_3's check request reaches N_2 after N_1 has informed N_2 of the outcome. This is a clear case, as N_2 at this point knows for certain whether R_2's molecule exists and it replies accordingly to N_3.

While R_2 could be executed *after* the consumption of its molecule, strong consistency is not needed for the execution of chemical programs. Indeed, the execution model provided by the chemical paradigm is implicitly parallel and non-deterministic. Therefore, there is no order of execution imposed on the execution runtime. Consequently, the succession of reactions can be reordered in an arbitrary manner. Seeing that N_3 has already captured the molecules it needs in the reaction *and* it has received a positive response to the existence of R_2's molecule, the execution can be regarded as one in which the order of N_1's and N_3's reactions has been inverted since in the chemical programming model, due to its non-determinism, the inversion does not influence the outcome of neither of the two reactions, and is thus considered correct. Note that cases with more nodes would naturally lead to the same conclusion. □

5.3.3 Inertia Detection

With rules being treated as usual, volatile molecules as well, another way of reaching inertia arises — one where there are no more rule molecules present in the solution. Thus, when a node commences a new execution cycle, its list of existent rules may be empty. In such a case, the node forces a recheck of all of the rules regardless of their recheck times, in order to be certain there are no more rule molecules present. If that is the case, it forces the recheck one more time before declaring inertia and terminating the execution, since a rule molecule might have appeared in the meantime. Naturally, if one or more rule molecules are found, the execution cycle continues.

Theorem 7. *When a node detects no more reactions can be performed, global inertia has been reached.*

Proof. As nodes do not collaboratively detect inertia, but individually, we have to prove there are no *false positive* detections. Consider the same network of nodes — N_1, N_2, N_3 — and program — R_1 and R_2 — as laid out in the proof of Theorem 6. Next, suppose N_1 does not perform a reaction, giving R_2's molecule back to N_2, Δt units of time after it has taken it. If during this period of time N_2 were not to send positive responses to check requests, N_3 would get a negative response to its check request. If, furthermore, R_2 were the only present rule N_3 knows of, it would falsely declare inertia, provided Δt were greater than the time it takes N_2 to recheck twice all of the rules. However, because it receives a positive response, N_3 continues the computation.

Another scenario where inertia might be detected falsely is when nodes try to determine whether there are still rule molecules present in the system. Let us assume there are three rules defined in the program: R_1, which consumes only data molecules; R_2, which creates an R_1 molecule; and R_3 which consumes two molecules: R_2 and R_3:

$$\begin{aligned}
&\textbf{let } R_1 = \textbf{replace } x :: gen, y :: gen \textbf{ by } x \\
&\textbf{let } R_2 = \textbf{replace } \omega \textbf{ by } \omega, x :: R_1 \\
&\textbf{let } R_3 = \textbf{replace } x :: R_2, y :: R_3, \omega \textbf{ by } \omega
\end{aligned}$$

Consider an execution environment composed of three nodes, N_1, N_2 and N_3, in the moment of execution where rule molecules of R_2 and R_3 are present in the solution, but not that of R_1. N_1 is about to determine whether inertia has been reached, N_2 is going to execute R_2, while N_3 is executing R_3. Suppose N_3 manages to complete its reaction during the period of time that N_1 is checking for R_1's existence. N_1 is going to get negative responses for all of the rules' molecules: for R_1 because it did not exist at the time, and for R_2 and R_3 because they were consumed in the meantime. Thus, as far as N_1 is concerned, inertia has been reached as it appears that there are no rule molecules left in the solution. However, while N_1 was checking for inertia, N_2 produced an R_1 molecule by executing R_2. N_1 is not going to declare inertia right away. Instead, it is going to do another rule-check round. It is going to discover the presence of R_1 in the solution, and is, consequently, not going to declare inertia.

Finally, nodes might falsely detect inertia because all of the rules are not available all of the time. Consider that two rules, R_1 and R_2, are defined in the program for which molecules for potential reactions exist. Let us suppose R_1 is available at the beginning of the execution, while R_2 appears dynamically after a period Δt. If the interval Δt is long enough for the system to declare all of the meta-molecules of R_2's reaction candidates as *inert*, then nodes could falsely detect inertia once R_1's candidates have been consumed. This, however cannot happen because the nodes in the *inert* part of the order-preserving layer are going to change the states of R_2' candidates' meta-molecules as soon as they detect the appearance of R_2's molecule in the system, in this way allowing its reactions' executions.

Given the fact that inconsistencies may arise only when rule molecules appear or are removed from the multiset, all of the possible cases of false inertia detection induced by them are covered above. Other, *"normal"*, cases of inertia detection have already been discussed in Theorem 5. □

5.4 Software Prototype

Following the descriptions of the platform, laid out in Sections 5.1 and 5.2, and its adaptation for accommodating the execution of higher-order programs, detailed in Section 5.3, a fully-functional prototype was developed in Java. Figure 5.6 shows its logical view.

5.4.1 Entities

It is composed of five entities:

Overlay Network. The abstraction from the underlying physical network is handled by this entity. Its main component is FreePastry, to which we added multi-threading support as each entity runs in its own thread.

Molecule Holder. This entity is the implementation of the uniform DHT layer and as such it serves as a container for molecules held by the node. In order to store, index and

retrieve molecules more easily, they are grouped by their molecule types and sorted based on their hash identifiers. The molecule holder is contacted during the atomic capture step and is in charge of deciding whether and to which node a molecule it holds is going to be given.

Meta-molecule Holder. Analogously, this entity represents the implementation of the order-preserving DHT layer and is, thus, a repository of meta-molecules. It manages the insertion, retrieval and deletion of meta-molecules requested by other nodes. Note that when a retrieval request is received, the meta-molecule is not removed. Instead, its copy is returned to the requesting node. Moreover, it handles random meta-molecule fetches and candidate requests. If it cannot satisfy the request, it communicates with meta-molecule holders on other nodes to complete it.

Tree Manager. The multicast tree created during the initialisation phase is constructed by this entity. It maintains the node's *local state* (consisting of its parent and children) and uses it to spread the rules down the tree and to send its and its children's remaining molecules to its parent.

Central Unit. This is the main entity in the prototype. It communicates with the application (taking the program to execute from it and returning the inert result to it) and executes the main execution loop (Algorithm 5.1). Furthermore, it is in charge of managing the *existent* and *non-existent* lists, *i.e.* it tracks the rules' existence.

Each of the entities cooperates with the entities directly above and below it portrayed in Figure 5.6.

5.4.2 Execution Cycle

5.4.2.1 Initialisation

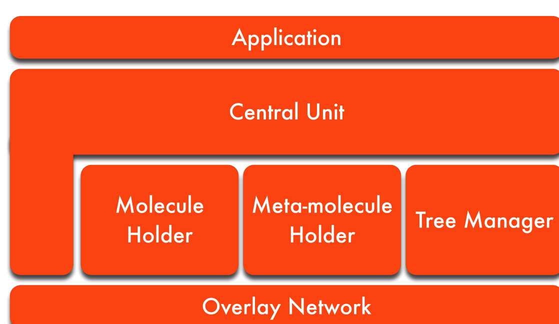


Figure 5.6: Logical view of the entities forming the prototype.

A first step is for the application to transfer the program to execute to the central unit. It then hashes the molecules and dispatches them to the overlay network, which spreads them in the uniform layer. During this period, the tree manager monitors the overlay network traffic and when it stumbles upon a molecule, it adds the destination node to its local state. Once the molecules have been disseminated, the central unit hands the definition of the rules over to the tree manager, which sends them to its children in the local state.

On the receiving end, when a node receives a molecule, it stores it in the molecule holder. This entity creates a meta-molecule for each held molecule and routes it in the order-preserving layer through the overlay network. The node then receives the rules to

execute, upon which the tree manager completes its local state by assigning the node's parent in the tree (the node which has sent it the rules). Now the nodes are ready to start the execution.

5.4.2.2 Execution

At this point, the central unit on each node starts the main execution loop by first looking whether there are rules in the non-existent list which have to be checked for existence, after which the active rule for the current execution cycle is selected. It then asks the meta-molecule holder to find it a random meta-molecule in the network. The type of one of the active rule's reactants is extracted. The meta-molecule holder then sends out requests in the *free* half of the order-preserving layer to find such a meta-molecule. This process is repeated for each rule until a meta-molecule has been returned. Then, the central unit *translates* the pair $(\mathtt{rule}, \mathtt{meta-molecule})$ into a range query request by injecting the meta-molecule's identifier in the rule. The request is, again, handed over to the meta-molecule holder which tries to find a candidate meta-molecule satisfying the range query in the order-preserving layer. The process of searching for a candidate is repeated for as many reactants the rule needs, each time introducing the newly acquired meta-molecule into the rule and constructing a new range query for the next candidate to be located. If the candidates cannot be found, the random meta-molecule's state is changed to *inert* (changing its identifier) and stored in the second DHT layer.

The final step of the execution phase consists in grabbing atomically the molecules and performing the reaction. For capturing the molecules the previously-described protocol is used, in which the central unit plays the role of the molecule requester. It extracts the identifiers of the molecules to grab and sends the fetch requests to the corresponding nodes. Their molecule holders then evaluate each its own request and decide whether to send back the molecules. Only in case all of the molecules have been received the node rechecks for the existence of the active rule. If the active rule is a higher-order one, *i.e.* if one or more rule molecules have been captured, the node sends a *DONE* message to the rule molecules' holders. Then, the reaction is performed and, as per Algorithm 5.1, the products of the reaction and their meta-molecules are stored in the DHT (each in their respective layer) and delete requests for the consumed molecules' meta-molecules are sent.

5.4.2.3 Termination

Once there are no more *free* meta-molecules in the order-preserving layer, the nodes enter the final, termination step of the execution. A node starts this phase when its meta-molecule holder is not able to find a random meta-molecule. At that point, the tree manager awaits the node's children's molecules. These are then combined with the molecules held by the molecule holder and sent to the parent up the tree. Finally, the central unit of the source node delivers the inert solution to the requesting application.

5.4.3 Optimisations

Even though the prototype follows the description of the system model discussed in earlier sections, it carries two slight improvements dealing with local meta-molecule search and meta-molecule retrieval. Both of the enhancements are implemented in the meta-molecule holder.

The first optimisation exploits the principle of *locality*: whenever the central unit requests for a meta-molecule, the meta-molecule holder first checks whether it can satisfy the request right away without querying other nodes. This method is applied to both random meta-molecule and candidate search requests. At the beginning of the execution, nodes which are located in the *free* half of the key space will be able to benefit directly from it, seeing that during that time most of the meta-molecules′ states are set to *free*, enabling nodes to pick a random meta-molecule from their local meta-molecule holder. Towards the end of the execution, on the other hand, nodes located in the other half of the key space can benefit from the principle during the candidate search step, since most meta-molecules will be labelled as *inert* at that point.

The second improvement is introducing a small decision-making mechanism into the meta-molecule holder. Whenever it receives a retrieval request, it tries to return the meta-molecule closest to the requested identifier. It is thus possible for the same meta-molecule to be sent to more than one node. Even though the capture protocol assures a molecule is going to be consumed in only one reaction, giving the same meta-molecule to different nodes generates superfluous network traffic as some grab requests will eventually be aborted. Therefore, the meta-molecule holder keeps track of the number of times each meta-molecule has been handed out to nodes. Doing so, it is able to return the meta-molecule which satisfies the request criteria but has been handed out less times than other meta-molecules. Such a slight refinement ultimately minimises the number of conflicts between nodes over molecules and consequently the network overhead due to capture aborts.

5.5 Evaluation

In order to evaluate our proposal, we tested the prototype described above on a real-world test-bed. The next section explains the test programs used in the evaluation, and Section 5.5.2 delivers the results of the conducted experiments.

5.5.1 Test Programs

We now describe three distinct classes of programs we used to evaluate the proposed architecture: highly-parallel, producer/consumer and higher-order ones.

5.5.1.1 Highly-parallel Programs

In such applications, the same operation is applied to the whole of the input data. It is thus interesting to investigate the behaviour of a decentralised execution environment when faced with such programs. We chose to represent them with `GetMax`, the program also used in the evaluation of the hierarchical prototype, described in Section 3.4.1.

5.5.1.2 Producer/Consumer Programs

The second category of applications is the producer/consumer one. The interest in this class stands in that these programs impose a certain degree of sequentiality of events — a producer has to produce the input for the consumer — on an implicitly parallel paradigm executing in a decentralised environment. While highly-parallel programs designate data-processing applications, producer/consumer ones can be interpreted as their temporal compositions — *workflows*.

The experiments were conducted with `StringManip` — a program comprised of two rules manipulating string molecules. The logic of `StringManip` consists in splitting and packing together string molecules in such a way that the resulting string molecules have a predefined length, denoted λ. The first rule, *SplitStr*, consumes one molecule whose string's length is greater than λ and produces two molecules; one is composed of the first λ characters of the original molecule's string, while the other contains its remainder. The second rule, *ConcatStr*, takes two molecules whose strings are shorter than λ characters as input and outputs one which is their concatenation. Thus, *SplitStr* produces the molecules which are going to be consumed by *ConcatStr*.

The course of the execution of `StringManip`, as well its outcome, is non-deterministic. While it is known that at the end of the execution the molecules' strings are going to have a length of λ, their contents depend on the succession of reactions performed by the system, which is influenced by the asynchronous nature of the platform. In other words, the outcome is conditioned by the reactions performed by each node, their input molecules, and the order in which they actually take place. Hence, the number of reactions done throughout the execution varies from run to run. Furthermore, the two rules are *circularly dependent* on each other — *ConcatStr* might produce molecules which are going to be consumed by *SplitStr* —, in this way bringing a partial *circular sequentiality* into the program.

5.5.1.3 Higher-order Programs

This type of programs contains higher-order rules, which are used by the program to regulate itself due to changes arising in its external environment. To assess the proposed platform's responsiveness to changes in the program's structure (its rules) induced by the dynamic addition and removal of rules, we use a simple program which first calculates the sum of some integers and then switches its computation to string concatenation based on integers currently present in the solution. The initial solution contains only integer molecules along with the *sum* rule, which takes two integers and produces their sum.

Then, at a random point in time, a new rule molecule is introduced — that of *switcher*, a rule erasing *sum*'s rule molecule — effectively ending the *sum epoch*. *switcher* produces two rule molecules: *int_to_str*, which turns integers into their string representations; and *str_concat*, which concatenates two string molecules.

5.5.2 Experimental Results

For conducting the experiments we were guided by three goals: *(i)* capturing the behaviour of the decentralised platform in terms of execution time and network traffic and comparing them to the results obtained analytically; *(ii)* assessing the overhead in execution time and network traffic created by the higher-order execution mechanisms from Section 5.3; and *(iii)* the platform's responsiveness when faced with dynamic changes in the program.

The experiments were conducted on Grid5000 [20], the French national test-bed. For each run, the nodes were spread randomly over nine geographically-distant sites connected via the RENATER network[1], which offers a 10 GBit/s communication channel between the sites. Each experiment was run 10 times and the figures given here represent the average values obtained during the experimentations.

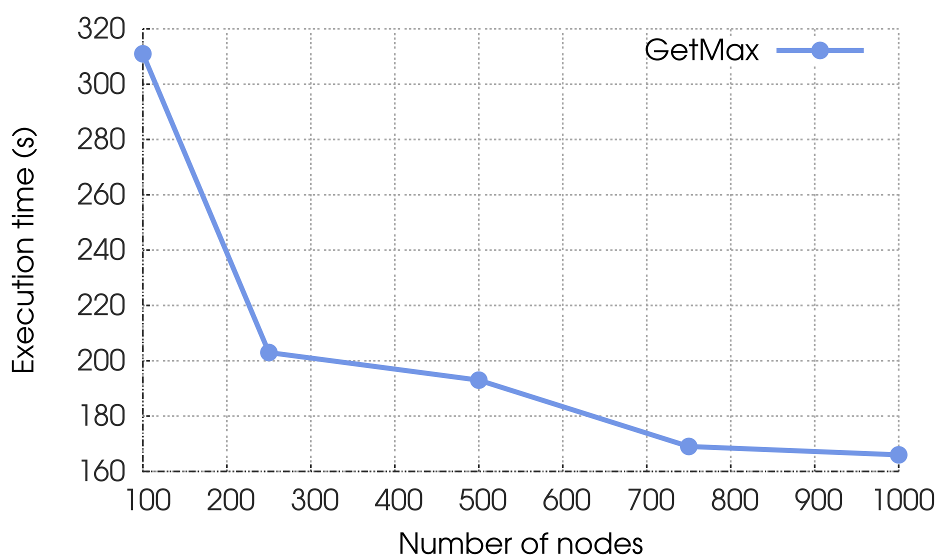


Figure 5.7: Execution time of `GetMax` containing 50,000 molecules.

5.5.2.1 Execution Time

Firstly, we evaluated the viability of the platform by executing the first two programs while varying the number of nodes participating in the execution. Figures 5.7 and 5.8 show the execution times obtained for `GetMax` and `StringManip`, respectively.

[1]`http://www.renater.fr`

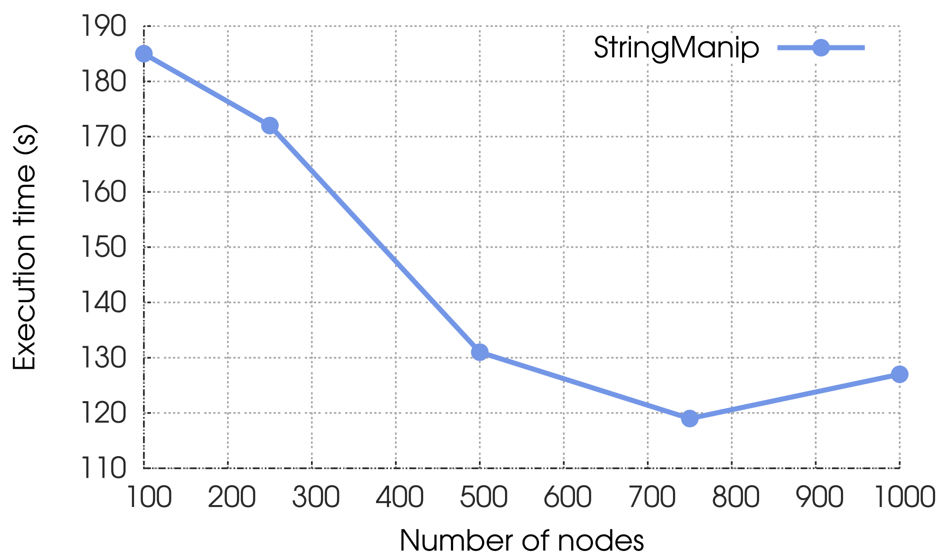


Figure 5.8: Execution time of `StringManip` with 20,000 molecules.

In both cases there is a decrease in execution time when increasing the number of nodes carrying out the computation, which is in compliance with the results of the complexity analysis from Section 5.2.5. However, one can notice that the speed-up obtained for `GetMax` is greater than that for `StringManip`. This is due to the difference of the programs' characteristics. On the one hand, the number of molecules in the system strictly decreases after each reaction when executing the `GetMax` program, while the trend is not known for `StringManip` — it may stay constant, decrease or increase. On the other, the execution time of `StringManip` depends on the sequentiality of events: certain reactions cannot be carried out before others are. In contrast, `GetMax` is a highly-parallel program where the maximum possible number of reactions can be performed at any given point in time. Finally, the execution takes more time to complete for 1000 nodes than for 750 when executing `StringManip`. This is the result of the program's sequentiality: more nodes are in conflict over a subset of molecules since not all available molecules can be used straight away, in this way prolonging the execution.

5.5.2.2 Network Traffic

During the execution of the programs we also monitored the generated network traffic. Figures 5.9 and 5.10 depict the total number of messages sent. Note that the number of messages in the case of `GetMax` is higher than that of `StringManip` due to the fact that there are more molecules in the initial solution of `GetMax`. Both of them show a linear augmentation in the number of messages, which conforms to the findings of the presented complexity analysis. We can see, however, that the curve for `GetMax` is steeper than that of `StringManip`. This is due to the fact that, because of the constant number of reactions, when there are more nodes involved in the computation there are more conflicts over molecules during the capture phase. In spite of this effect, one can notice that the actual

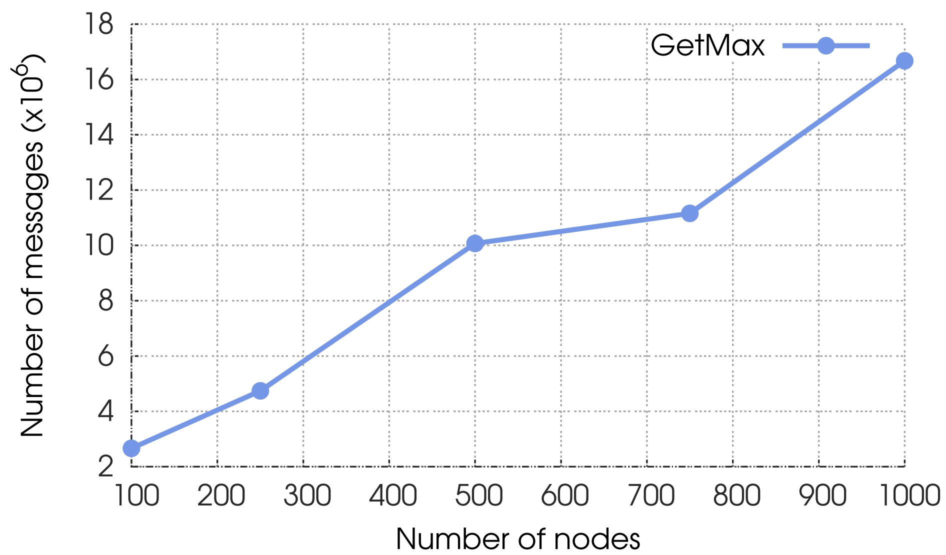


Figure 5.9: Number of messages sent during the execution of `GetMax`.

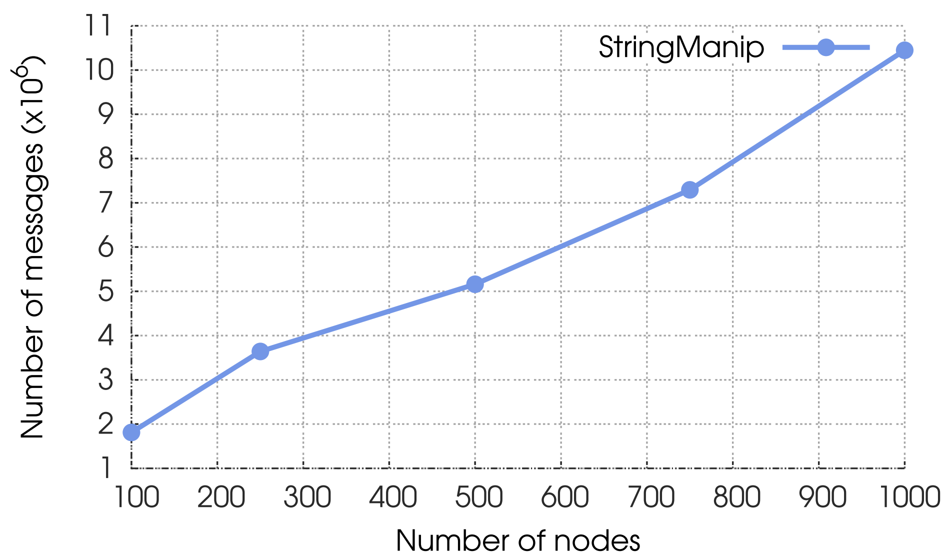


Figure 5.10: Number of messages sent during the execution of `StringManip`.

number of messages per node declines with the growth of the network, which leads to the conclusion that the platform is scalable in terms of network load. We can thus conclude that the platform scales well.

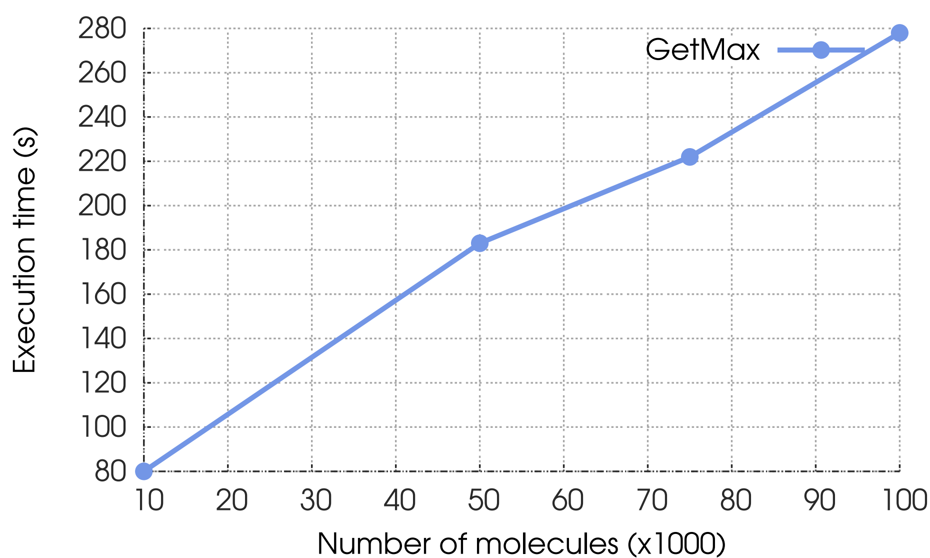


Figure 5.11: Execution time of `GetMax` on 500 nodes.

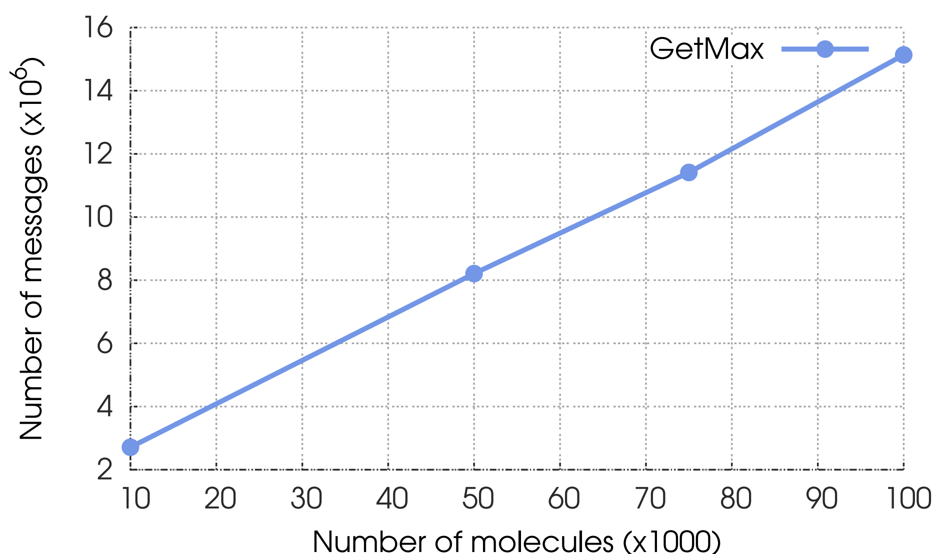


Figure 5.12: Number of messages sent during the execution of `GetMax`.

5.5.2.3 Problem Size Variation

In this experiment we fixed the number of nodes to 500 while varying the number of molecules contained in the `GetMax` program. Figure 5.11 shows that the execution time

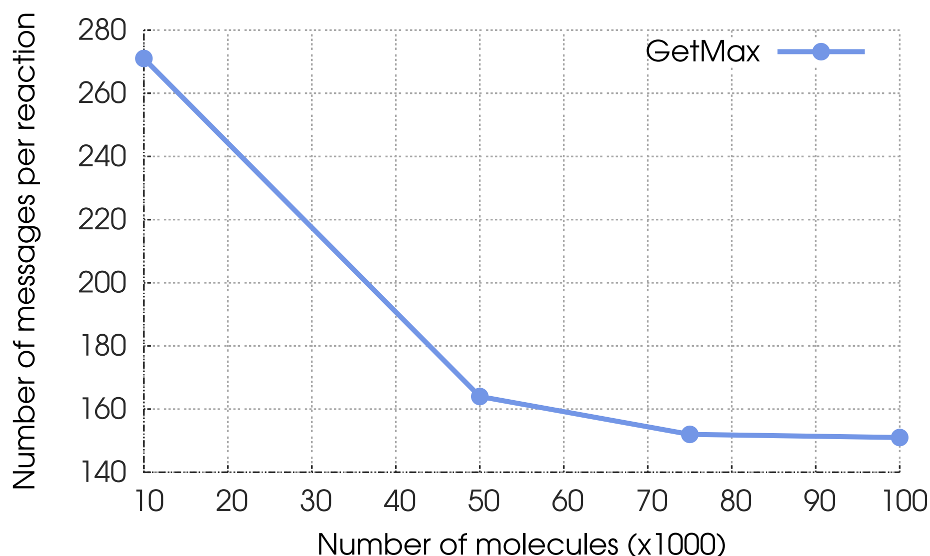


Figure 5.13: Number of messages sent per reaction during the execution of `GetMax`.

linearly grows with the increase of the size of the problem. The same effect can be observed when looking at the network traffic, depicted in Figure 5.12. Both figures confirm the analysis' findings: the number of messages linearly grows when increasing the size of the problem to be solved, *i.e.* the number of molecules. It is interesting to note that the number of messages needed to perform one reaction, illustrated in Figure 5.13, decreases. Indeed, when the number of molecules increases while keeping the number of nodes constant, there are less conflicts between nodes over molecules, and thus less communication is needed to carry out a reaction.

5.5.2.4 Higher Order Overhead

We now turn to the experiments concerning the higher-order mechanisms. We firstly investigated the overhead induced by the higher-order mechanisms by executing the `GetMax` program in a network of 250 nodes while varying the size of the problem, *i.e.* the number of molecules contained in the initial solution. The duration and the network traffic are shown on Figures 5.14 and 5.15, respectively. One can firstly note the linear behaviour of the curves, confirming that the platform keeps the property of linearity even with the higher-order mechanisms activated.

When comparing the behaviour of the system with and without higher-order mechanisms in place, we can observe that when the size of the problem is relatively small, the difference in both execution time and network traffic is negligible. However, the gap widens with the augmentation of the problem's size. This trend is to be expected, since the higher-order mechanism *inserts* the rule-check procedure between the molecule capture and reaction execution steps, in this way both prolonging the execution and creating additional network traffic. In spite of this penalty, the platform is still able to follow the linear trend of the growth in size.

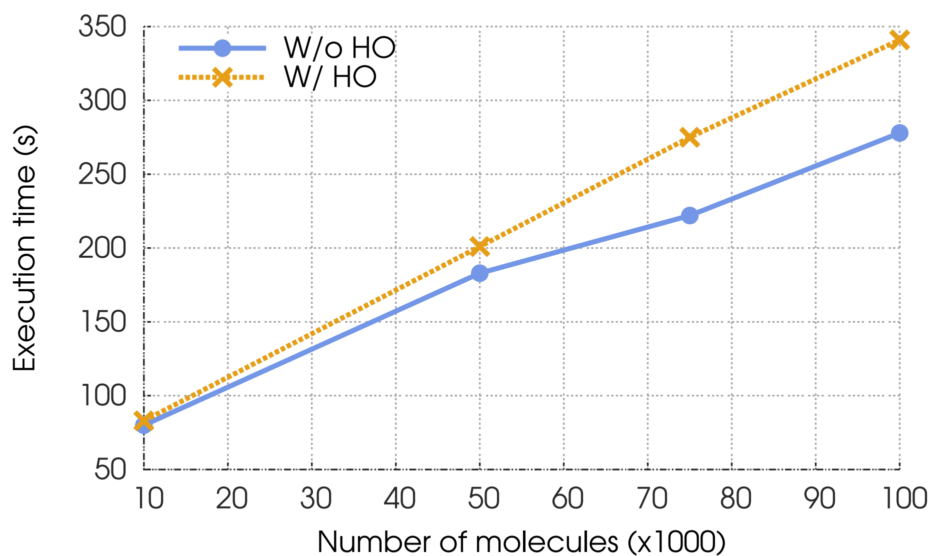


Figure 5.14: Execution time of `GetMax` when varying the number of molecules, with and without activating the higher-order mechanism.

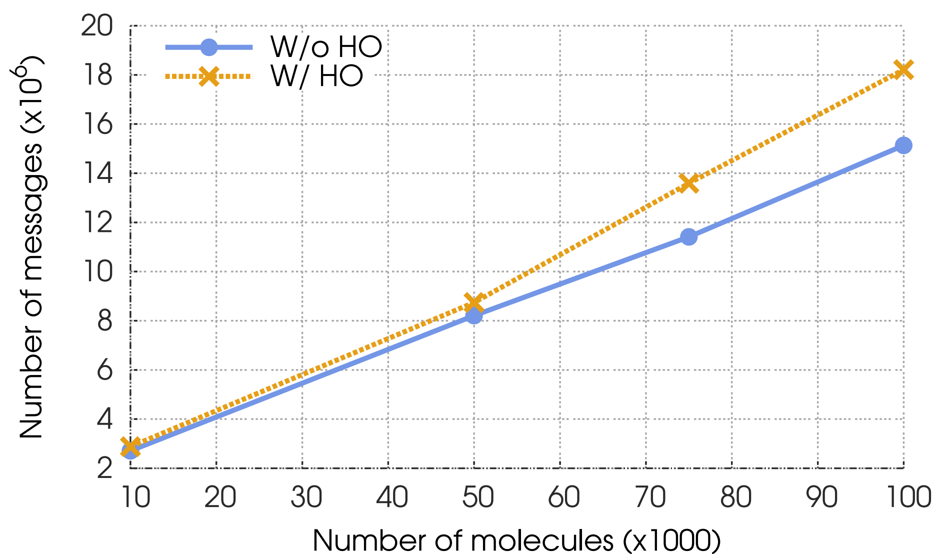


Figure 5.15: Number of messages exchanged by nodes during the execution of `GetMax` when varying the number of molecules, with and without activating the higher-order mechanism.

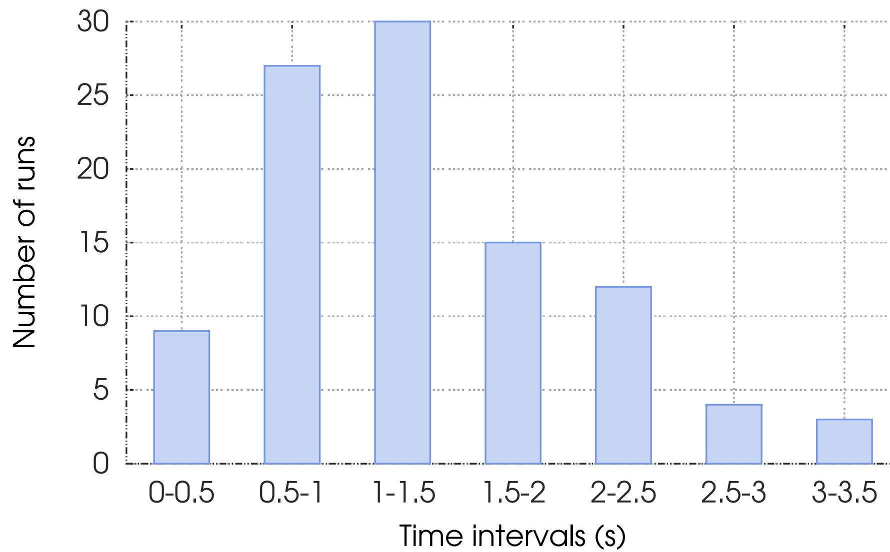


Figure 5.16: Responsiveness of the platform with regards to the time it takes to detect the disappearance of a rule.

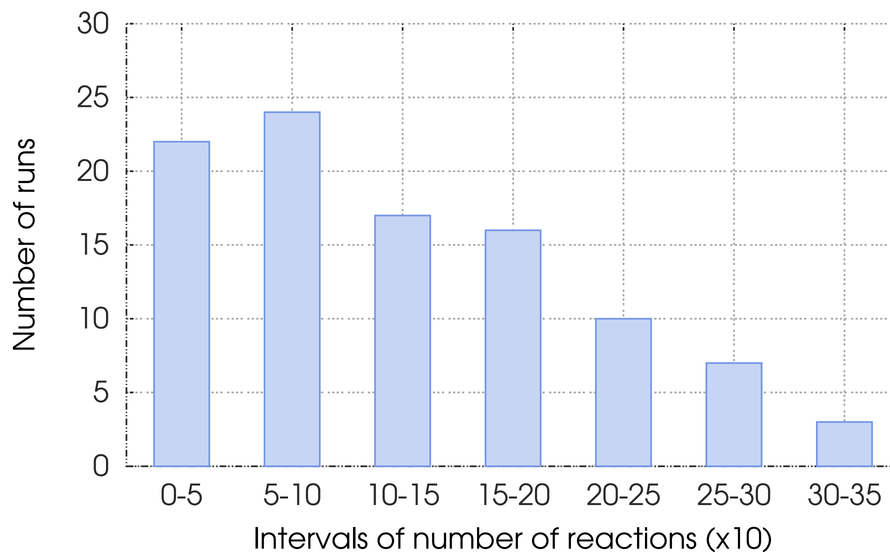


Figure 5.17: Responsiveness of the platform in terms of number of reactions done with a non-existing rule.

5.5.2.5 Responsiveness to Dynamics

In the last experiment we used the higher-order program described above in order to assess the platform's responsiveness to dynamic changes in the program's execution flow. We executed the program 100 times on a network of 250 nodes, changing the network configuration each time by randomly choosing the site, the cluster and the port number of each node. We measured the time it took to all of the nodes to *realise* that the *sum* rule's molecule is not present in the solution any more. At the beginning of each run all of the nodes' internal clocks were synchronised through the NTP service provided by Grid5000 in order to obtain precise measurements. Figure 5.16 shows the number of runs in which the nodes stopped executing the rule after its disappearance from the solution, divided in half-second intervals. Likewise, Figure 5.17 shows the number of reactions carried out during that period.

As explained in Section 5.3, this delay manifests itself because *sum*'s molecule's holder replies positively to rule check requests until it has received the confirmation of its consumption, and it, thus, depends on the underlying network's topology. Based on Figure 5.16, we observe that in 81% of runs the nodes stop executing the rule at most two seconds after its disappearance, while the highest delay is 3.5 seconds, observed in only 3% of runs. Even though this might seem as a long period of time, the two-second delay represents only 1% of the overall execution time for the given scenario. In the same vein, Figure 5.17 suggests that in 79% of runs nodes do at most 200 reactions during the delay, which represents 1% of the total number of reactions done during the execution and less than one reaction per node. We can, thus, conclude that the platform reacts quickly to changes, both in terms of time it takes it to detect the change and in terms of number of reactions which should have taken place *before* the change in a sequential ordering of events.

5.6 Conclusion

Continuing the work started in Chapter 4, this chapter presents a fully-decentralised runtime for the execution of chemical programs in large-scale environments.

While ensuring the atomic capture of reactants is essential to a decentralised system, the chemical programming model introduces further challenges. By relying on a double DHT layer, the presented platform is able both to secure scalable inter-node communication and to provide an efficient inertia detection mechanism. Molecules are spread uniformly around the original DHT ring, in this way balancing access to them. The platform maintains a second DHT layer where meta-molecules — molecule descriptors — are placed. By ordering this layer in an order-preserving manner, the system can use range queries to locate specific molecules matching a pattern, in this way providing advanced reactant search capabilities. Once located through their meta-molecules, molecules are fetched in an atomic fashion using the adaptive capture protocol described in the previous chapter. Finally, the platform has the ability to execute higher-order programs: it periodically checks for the existence of the defined rules' molecules to insure consistency and

a correct execution. The properties of these concepts are theoretically analysed, while inertia detection and execution correctness are formally proven.

This chapter also introduces the prototype built, constituting a first step towards the generic execution of chemical programs in large-scale environments. Apart from being modelled on the theoretical work described, it also incorporates a couple of optimisations, which improve its scalability as they exploit thelocality of meta-molecules. The prototype has been used to conduct experiments on a real-world large-scale test-bed. The results corroborate the theoretical findings and show the platform's viability and scalability.

Part III

Conclusions And Addenda

Conclusion

In the context of distributed systems, autonomic computing envisions self-manageable systems, *i.e.* systems which are able to organise, configure, heal and protect themselves. However, one of the difficulties of today's distributed systems stands in finding the right abstractions to program them. Due to their complexity in terms of scale, dynamics and heterogeneity, leveraging their power remains an open issue. In particular, separating the logic of a computation (*"what we want to do"*) from the low-level, technical implementation (*"how to do it"*) appears to be a prerequisite for being able to compute on such systems in an efficient manner. Rule-based programming seems a likely candidate for this separation as in such a paradigm, the logic is expressed as a set of high-level rules, hiding the intrinsic difficulties of parallelism and distribution inherent to real platforms.

In this thesis, we focus on a higher-order rule-based model, namely the chemical programming model. It associates rule-based programming with an implicitly-parallel runtime. It also integrates the higher-order, *i.e.* the ability to manipulate rules as ordinary objects, and thus to modify the program dynamically at run time. As the paradigm has been shown to have the right abstractions for the specification of autonomic systems, the large-scale execution of chemical programs has to be tackled in order to put the attractive characteristics of rule-based programming into practice over large-scale platforms. This thesis fills this gap by proposing a generic framework to solve this issue over a largely-distributed platform. By devising an adaptive and efficient protocol for atomically capturing molecules and designing an efficient inertia detection mechanism, we modelled a decentralised chemical runtime able to run over large-scale platforms.

The thesis is composed of two parts: the first reviews the preliminaries needed to understand our contribution, while the second describes it in detail. In Chapter 1 we explained the chemical programming model, concentrating on its execution properties, such as non-determinism, implicit parallelism and inertia detection. There is also a detailed description of the principles behind distributed hash tables and the way more complex queries, such as range queries, may efficiently be resolved over them, since we used them as the basis for the inter-node communication and search mechanisms, respectively.

The last section gave a brief overview of the mutual exclusion algorithms, as they were needed in order to formulate our own capture protocol. Chapter 2 highlighted the need for a generic runtime by analysing different existing proposals of distributing the execution of chemical programs, both based on message-passing as well as those exploiting the shared-memory model.

Due to the lack of an appropriate distributed execution framework, in Chapter 3 we described a study conducted about the feasibility of a generic runtime able to run any chemical program on any platform. To that end, we explored two approaches: one based on the shared-memory model, and the other on message passing. The former exploits the concept of distributed shared memories to create a collective virtual space from which each node picks combinations of molecules to perform reactions. After a careful analysis of the drawbacks and inherent limitations of the approach, we abandoned it and turned to message passing as the underlying type of communication. Using the Pastry distributed hash table as the underlying communication layer, we managed to build a runtime able to run chemical programs in large-scale environments. Here we were primarily concerned with the distribution of the execution process and the mechanism allowing the runtime to detect inertia in a distributed manner, and proposed a hierarchical, tree-based execution scheme within which operates a minimal, distributed inertia-detection algorithm. Due to the hierarchical structuring of the model, we examined its load-balancing properties and formulated a tree-reorganisation scheme transferring part of the execution from parent to child nodes. A fully-functional prototype was developed based the execution-tree approach and tested on a real-world, large-scale platform. The results of the experiments confirmed the viability and scalability of such a platform and corroborated the minimality of the inertia-detection algorithm obtained through formal analysis.

On the road towards a decentralised chemical runtime, in Chapter 4 we tackled the problem of mutual exclusion in the context of molecule capture. The contribution here is an adaptive, efficient and scalable protocol for the *atomic* capture of molecules in large-scale systems. It works in two regimes — optimistic and pessimistic — with an efficient switch mechanism based on local capture histories. The activation of each mode follows the concentration of molecules, and thus the reaction potential, in the system. When an elevated number of molecules is present, nodes use the optimistic sub-protocol to grab molecules. While it is the faster and lighter of the two sub-protocols in terms of communication costs, it is not able to ensure liveness when the number of conflicts over molecules is high. Therefore, when the number of molecules in the system drops, nodes switch to the pessimistic sub-protocol, which, even though heavier and costlier, is able to guarantee liveness is preserved. To complete the protocol, we also proposed in this chapter a rule-changing mechanism which instructs the nodes as to which rule to execute in a multi-rule program. The formal proof of the protocol is also provided, and the conducted simulations revealed the protocol's efficiency, low overhead and ability to quickly react to changes, both with regard to the switch between the two sub-protocols as well as to picking the rule to execute.

Finally, in Chapter 5 we explained the characteristics of our decentralised runtime for the execution of chemical programs at large scale, built around the aforementioned cap-

ture protocol. Constructed atop the Pastry DHT, the runtime features two DHT layers — one uniform, the other order-preserving — bringing intelligent search capabilities into the picture. Unlike the hierarchical runtime, where all of the molecule combinations must be examined, in this runtime nodes are able to search for precisely the molecules they need in order to perform reactions. If found, the sought molecules are then grabbed using the capture protocol. Inertia is detected in an efficient and fully-decentralised way due to the molecule classification scheme built into the double layer. The runtime is also able to execute higher-order programs, in this way allowing the programs to dynamically change throughout their execution. In the chapter we also presented the prototype built. Apart from following the laid-out model of the runtime, it features some optimisations based on the principle of locality, which improve the framework's execution time and decrease the network traffic. The tests performed in a large-scale test-bed with up to 1000 nodes on highly-parallel and sequential chemical programs showed the runtime's viability and scalability, as well as its ability to quickly detect changes when higher-order programs were run.

Future Work

Even though this thesis gives a complete solution for running chemical programs over large-scale platforms in a fully-decentralised manner, there is room for improvement in several directions. The work to be undertaken can be divided in two categories : engineering and research.

Engineering Challenges

The prototype for the decentralised chemical runtime developed during the thesis is fully functional in the sense that it can be used to test programs following the chemical paradigm in large-scale environments. Nevertheless, it is still only a prototype. More engineering effort has to be put into it in order to turn it into a real *chemical middleware*. Some of the most significant improvements needed follow.

Compiler Integration. In its current state, the prototype is only able to execute programs *following the chemical paradigm*, but cannot execute HOCL code directly. In order words, the user has to *translate* their program manually to execute it over a large-scale platform. Therefore, the first step towards a real middleware must be the integration of the HOCL compiler into the prototype. This process should be straightforward because the object model used in the prototype was carefully chosen to resemble as much as possible that of the existing HOCL compiler.

Rule Translator. As outlined in Chapter 5, the intelligent search for candidates is based on the rule translator — an entity which modifies the reaction condition by inserting the found meta-molecules in order to establish the range of values where the next reaction candidate can be found. In the current version of the prototype, the

rule translator is regarded as a black box; its functionality is provided by the user through some additional methods supplied with the definition of a rule. The realisation of an *automated* rule translation ought to be easy to accomplish after the integration of the compiler since: *(i)* the automation of the process requires an HOCL parser, which is an integral part of the compiler; and *(ii)* the range query mechanisms and the order-preserving layer are, in a sense, independent from other prototype's functionalities, allowing the easy modification of the rule translator.

Range Query Optimisation. Range queries are performed in order to locate potential reaction candidates. However, it being a proof of concept, the prototype uses a basic, MAAN-like mechanism for carrying them out. There is, thus, room for improving the performance of this step, both in terms of time and network traffic. As, again, these range query mechanisms are isolated from other parts of the prototype, implementing a different, optimised version, such as Squid or inverted space-filling curves, should be a fairly straightforward task.

Load Balance. Currently, all of the nodes participate equally in all of the aspects of the execution; all of them try to carry out reactions, search for candidates, as well as store molecules and meta-molecules. In a heterogeneous environment, however, not all of the nodes might be able to perform all of the functions. As an example, if one of the participating nodes is run on a mobile phone, it should be assigned a reduced workload. To take this heterogeneity into account, different profiles and configurations dependent of the underlying resources could be developed. This would benefit both the platform (as it would not overstress *weaker* nodes) and its users (as they could continue working without interruptions).

Research Directions

Fault Tolerance. One of the most important aspects lacking in the design of the decentralised runtime is resilience to crashes. Since it is meant to be used in large-scale platforms, where faults do not represent the exception but the rule, the runtime must be able to handle these cases gracefully and ensure the continuation of the computation in spite of the possible problems arising in the underlying platform. On the lower level, the usage of distributed hash tables provides rudimentary fault tolerance in the sense that in case of crashes, the communication pattern of the runtime is going to be unaffected. However, a higher-level mechanism is needed in order to ensure that no disruption in execution will occur. While the order of the succession of reactions is chosen non-deterministically, the execution steps of each node are performed in a deterministic and *a priori* known order. Therefore, they can be described as finite-state automata, for which resilience can be provided in the form of state-machine replication [78, 112]. Each node could replicate its whole state onto a number of its neighbours. In this way, in case it crashes, another node may simply overtake its part of the computation, increasing the robustness of the runtime. Apart from *complete,* fail-stop crashes, the system is also susceptible to other types of failures, namely transient (where a node stops working for a short period of time)

and Byzantine ones (where malicious nodes try to destabilise the system). In the case of transient faults, self-stabilisation techniques [38] are worth exploring, as they can guide the system towards a correct and coherent state during the faulty period. In a more general sense, however, given the specificities of the chemical paradigm's execution model, it would be interesting to determine the weakest failure detection model [27] able to support, primarily, the loss of molecules.

Security. Another research direction tightly connected to large-scale environments is security. It is clear that any system aspiring to be used in the real world must guarantee a secure environment to its users. Currently, our runtime does not deal with such issues, as it assumes the collaborating nodes and their users to be benign. In order to provide a safe and healthy environment in which users can freely run their chemical applications, the runtime must provide security on three levels:

- *Environment Level*. Each node must be certified prior to joining the runtime for the first time. The certification could be done by a platform administrator explicitly allowing each of them to participate in the execution or by handing out runtime-specific certificates.
- *User Level*. If a user wants to execute a chemical application, they should be allowed to do so only if they have been granted *a priori* permission, precluding malicious users from abusing the runtime. Likewise, known users could be monitored in order to ensure a fair usage of the execution environment.
- *Application Level*. Even though users are trusted, applications which they submit could, voluntarily or not, contain malicious code. Since the runtime executes directly HOCL code and not its pre-compiled version, it could incorporate code-verification schemes able to check if the application could perform unauthorised actions. However, due to the dynamic, higher-order nature of chemical programs, this might be hard to achieve; not only does the higher order allow programs to change their execution flow dynamically, but it also enables them to *alter* the code being executed. In this sense, an attractive research direction seems to be on-line code verification: first, the code could be entirely checked prior to its execution, and then, when a dynamic change is detected, another verification cycle could be initiated. Alternatively, applications could be *sandboxed, i.e.* they might be executed on the participating nodes in an untrusted environment. In case a security breach occurs, the application's processes on the nodes could be simply terminated.

Applications. Finally, the modelled runtime should be tested not only in real-world conditions but also with real-world applications, preferably autonomic systems running on top of real-world large-scale platforms, in order to fully assess the benefits and drawbacks of the proposed runtime. As suggested in the introduction of this thesis, one of the envisioned directions is workflow management and execution. Some work has already been

done in that regard [40]. While it gives a complete specification for the decentralised enactment, execution and management of scientific workflows using the chemical programming paradigm, the work presented in [40] provides only a partially-distributed execution environment which was used for testing purposes only. Taking into account this chemical specification, we have adapted our runtime for the decentralised scheduling of multiple workflows [TR-7925]. The conducted simulations show the adaptation to be promising. However, due to the lack of time, we haven't had the chance yet to perform experiments in a real-world setting.

Alternative Approaches. In this thesis, we focused on developing a decentralised runtime atop structured peer-to-peer networks, namely distributed hash tables. Yet, other network-organisation schemes might be equally suitable for supporting the generic execution of chemical programs. Due to the scale of the platforms targeted, an alternative approach could be modelling a runtime using population protocols [8]. There, interactions happen on a one-to-one basis between nodes and their formal model secures all pairs of nodes will eventually communicate, in this way ensuring the convergence of the calculation. The randomised nature of node interactions represents some similarity with the non-determinism witnessed in the chemical programming paradigm, and can be, thus, used to *naturally* model molecule encounters which produce reactions. However, due to the one-to-one communication used in population protocols, alternative molecule discovery algorithms would have to be devised. In other words, the following question ought to be answered: how to perform intelligent searches for candidates in a structureless network in which the basis of communication are one-to-one interactions? Analogously, an auxiliary inertia detection mechanism would have to be formulated. On the practical side, an implementation of such a runtime could be based on gossip protocols [59] because of their similarity with population protocols [18].

In a wider context, the work presented in this thesis is part of an ambitious plan to use the chemical programming paradigm to model self-manageable, large-scale platforms. The paradigm inherently equips the user with self-adaptation capabilities, and previous works, such as [40] and [99], provide them with the concepts needed to ensure properties such as self-coordination. This thesis fills the gap between the conception and actual execution of chemical programs, which represents a step required in the way towards next-generation middleware.

Bibliography

[1] Amazon's outage in third day: debate over cloud computing's future begins. http://venturebeat.com/2011/04/23/amazons-outage-in-third-day-debate-over-cloud-computings-future-begins/. Retrieved online on Dec 10, 2012. 1

[2] Risk of mass electricity blackouts may still be rising. http://venturebeat.com/2011/04/23/amazons-outage-in-third-day-debate-over-cloud-computings-future-be Retrieved online on Dec 10, 2012. 2

[3] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A Rule-based Language for Web Data Management. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS'11)*, pages 293–304, Athens, Greece, 2011. 5, 171

[4] D. Agrawal and A. El Abbadi. Efficient solution to the distributed mutual exclusion problem. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 193–200. ACM, 1989. 40

[5] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Second International Conference on Peer-to-Peer Computing*, pages 33–40. IEEE, 2002. 37

[6] F. Araújo and L. Rodrigues. Survey on distributed hash tables. 2006. 29

[7] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997. 37

[8] J. Aspnes and E. Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, 2007. 158

[9] J. Aspnes and G. Shah. Skip graphs. *ACM Transactions on Algorithms (TALG)*, 3(4):37, 2007. 37

[10] R. Baldoni. An o($n^{M/(M+1)}$) distributed algorithm for the k-out of-m resources allocation problem. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 81–88. IEEE, 1994. 45

[11] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Gener. Comput. Syst.*, 4:133–144, September 1988. 9, 20, 21, 50, 52, 171, 173, 183

[12] J.-P. Banâtre, P. Fradet, and Y. Radenac. Higher-order chemical programming style. *Proceedings of the Workshop on Unconventional Programming*, 3566:84–95, 2005. 22

[13] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised Multisets for Chemical Programming. *Mathematical Structures in Computer Science*, 16, 2006. 10, 174

[14] J.-P. Banâtre, P. Fradet, and Y. Radenac. Towards Chemical Coordination for Grids. In *SAC*, 2006. 10

[15] J.-P. Banâtre and T. Priol. Chemical Programming of Future Service-oriented Architectures. *Journal of Software*, 4, 2009. 171, 174

[16] Y. Bar-Yam. *Dynamics of Complex Systems*. Studies in Nonlinearity. Westview Press, July 2003. 2

[17] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981. 56

[18] M. Bertier, Y. Busnel, and A.-M. Kermarrec. On gossip and populations. *Structural Information and Communication Complexity*, pages 72–86, 2010. 158

[19] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 353–366. ACM, 2004. 37

[20] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Winter 2006. 83, 142, 184

[21] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004. 37

[22] K. S. Candan, J. Tatemura, D. Agrawal, and D. Cavendish. On Overlay Schemes to Support Point-in-range Queries for Scalable Grid Resource Discovery. *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005. 129

[23] J. Cardoso, A. Barros, N. May, and U. Kylau. Towards a unified service description language for the internet of services: Requirements and first developments. In *IEEE International Conference on Services Computing (SCC)*, pages 602–609. IEEE, 2010. 3

[24] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, page 164. ACM, 1991. 68

[25] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, 1983. 40

[26] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002. 70, 71

[27] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996. 157

[28] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984. 42, 183

[29] B. Chazelle. Computational geometry on a systolic chip. *IEEE Transactions on Computers*, 100(9):774–785, 1984. 54

[30] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Multidimensional voting: A general method for implementing synchronization in distributed systems. In *10th International Conference on Distributed Computing Systems*, pages 362–369. IEEE, 1990. 40

[31] C. Creveuil. Implementation of gamma on the connection machine. *Reasearch Directions in High-Level Parallel Programming Languages*, pages 219–230, 1992. 53, 54

[32] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 57–66. IEEE, 2005. 37

[33] N. G. de Bruijn and P. Erdos. A combinatorial problem. *Koninklijke Netherlands: Academe Van Wetenschappen*, 49:758–764, 1946. 35

[34] C. Di Napoli, M. Giordano, J.-L. Pazat, and C. Wang. A Chemical Based Middleware for Workflow Instantiation and Execution. In *ServiceWave*, pages 100–111, 2010. 10, 171

[35] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1(2):115–138, 1971. 41

[36] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries – a Review. *Artificial Life*, 7:225–275, June 2001. 171

[37] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey. Fulfilling the vision of autonomic computing. *Computer*, 43(1):35–41, 2010. 4

[38] S. Dolev. *Self-stabilization*. MIT press, 2000. 157

[39] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1), 2001. 29, 33

[40] H. Fernández. *Flexible Coordination through the Chemical Metaphor for Service Infrastructures*. PhD thesis, Université de Rennes 1, 2012. 158

[41] H. Fernández, T. Priol, and C. Tedeschi. Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In *IEEE ICWS*, 2010. 10, 171, 184

[42] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254. IEEE, 1979. 42

[43] P. Fraigniaud and P. Gauron. D2b: A de bruijn based content-addressable network. *Theoretical Computer Science*, 355(1):65–79, 2006. 35

[44] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the Thirtieth international conference on Very large data bases*, volume 30, pages 444–455. VLDB Endowment, 2004. 37

[45] J. Gao and P. Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *12th IEEE International Conference on Network Protocols*, pages 239–250. IEEE, 2004. 38

[46] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM (JACM)*, 32(4):841–860, 1985. 40

[47] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *W. H Freeman Press: USA*, 1979. 28

[48] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Not.*, 45(3):347–358, March 2010. 65

[49] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979. 40

[50] K. Gladitz and H. Kuchen. Shared memory implementation of the gamma-operation. *Journal of Symbolic Computation*, 21(4):577–591, 1996. 56

[51] S. Grumbach and F. Wang. Netlog, a Rule-Based Language for Distributed Programming. In *12th International Symposium on Practical Aspects of Declarative Languages*, pages 88–103, 2010. 5, 171

[52] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394. ACM, 2003. 29

[53] C. Hankin, D. Le Métayer, and D. Sands. A parallel programming style and its algebra of programs. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Parallel Architectures and Languages Europe (PARLE)*, volume 694 of *Lecture Notes in Computer Science*, pages 367–378. Springer Berlin / Heidelberg, 1993. 27

[54] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, volume 4, pages 9–9, 2003. 37

[55] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001. 2

[56] L. Huang, W. Tong, W. N. Kam, and Y. Sun. Implementation of gamma on a massively parallel computer. *Journal of Computer Science and Technology*, 12(1):29–39, 1997. 53, 54, 183

[57] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995. 10

[58] Y. Jégou. Dynamic Memory Management on Mome DSM. *Cluster Computing*, 2006. 68

[59] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007. 158

[60] Y. J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000. 42

[61] M. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table. *Peer-to-Peer Systems II*, pages 98–107, 2003. 36

[62] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. *IEEE Transactions on Computers*, 42(5):553–558, 1993. 43

[63] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters*, 49(4):213–218, 1994. 43

[64] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997. 32

[65] J.R. Karr, J.C. Sanghvi, D.N. Macklin, M.V. Gutschow, J.M. Jacobs, B. Bolival, N. Assad-Garcia, J.I. Glass, and M.W. Covert. A whole-cell computational model predicts phenotype from genotype. *Cell*, 150(2):389–401, 2012. 2

[66] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. 2

[67] D. E. Knuth. *The art of computer programming*. addison-Wesley, 2006. 53

[68] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, 1991. 40

[69] Z. Laliwala, R. Khosla, P. Majumdar, and S. Chaudhary. Semantic and Rules Based Event-Driven Dynamic Web Services Composition for Automation of Business Processes. In *Services Computing Workshops (SCW '06)*, pages 175–182, 2006. 5

[70] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 40, 92, 172

[71] B. Lampson and H. Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center, 1979. 179

[72] P. Langendoerfer, K. Piotrowski, M. Diaz, and B. Rubio. Distributed shared memory as an approach for integrating wsns and cloud computing. In *5th International Conference on New Technologies, Mobility and Security*, pages 1 –6, may 2012. 65

[73] D. Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), 1998. 10

[74] A. Lèbre, R. Lottiaux, E. Focht, and C. Morin. Reducing kernel development complexity in distributed environments. In *Euro-Par 2008*, volume 5168 of *Lecture Notes in Computer Science*, pages 576–586. 2008. 68

[75] H. Lin, J. Kemp, and P. Gilbert. Computing Gamma Calculus on Computer Cluster. *International Journal of Technology Diffusion (IJTD)*, 1(4):42–52, 2010. 183

[76] J.W. Lloyd. Practical Advtanages of Declarative Programming. In *Joint Conference on Declarative Programming (GULP-PRODE'94)*, pages 18–30, 1994. 4, 171

[77] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, pages 72–93, 2005. 29, 34

[78] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in distributed hash tables. pages 295–305, 2002. 71, 94, 156

[79] M. Maekawa. An sqrt(n) algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2):145–159, 1985. 40, 43, 44

[80] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192. ACM, 2002. 35

[81] Y. Manabe, R. Baldoni, M. Raynal, and S. Aoyagi. k-arbiter: A safe and general scheme for h-out of-k mutual exclusion. *Theoretical Computer Science*, 193(1):97–112, 1998. 45

[82] Y. Manabe and N. Tajima. (h,k)-arbiters for h-out-of-k mutual exclusion problem. *Theoretical computer science*, 310(1):379–392, 2004. 45

[83] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002. 36

[84] D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM Sigmod Record*, 18(2):215–224, 1989. 4

[85] H. McEvoy. *Gamma, chromatic typing and vegetation*. Imperial College Press, 1996. 10

[86] D. Mentré, D. Le Métayer, and T. Priol. Formalization and verification of coherence protocols with the gamma framework. In *PDSE*, 2000. 10

[87] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing, 2002. 29

[88] M. Mizuno, M. L. Neilsen, and R. Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. In *11th International Conference on Distributed Computing Systems*, pages 361–368. IEEE, 1991. 41

[89] A. Mostéfaoui. Towards a Computing Model for Open Distributed Systems. In *9th International Conference on Parallel Computing Technologies (PaCT)*, pages 74–79, 2007. 10, 171

[90] M. Naimi and M. Trehel. How to detect a failure and regenerate the token in the log (n) distributed algorithm for mutual exclusion. *Distributed Algorithms*, pages 155–166, 1988. 41

[91] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *11th International Conference on Distributed Computing Systems*, pages 354–360. IEEE, 1991. 41

[92] M. L. Neilsen, M. Mizuno, and M. Raynal. A general method to define quorums. In *12th International Conference on Distributed Computing Systems*, pages 657–664. IEEE, 1992. 41

[93] Z. Németh, C. Pérez, and T. Priol. Distributed workflow coordination: molecules and reactions. In *IPDPS*, 2006. 10

[94] S. Nishio, K. F. Li, and E. G. Manning. A resilient mutual exclusion algorithm for computer networks. *Parallel and Distributed Systems, IEEE Transactions on*, 1(3):344–355, 1990. 41

[95] M. E. O'Neill. The genuine sieve of eratosthenes. *Journal of Functional Programming*, 19(1):95, 2009. 21

[96] M. Parashar and S. Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, pages 97–97, 2005. 2, 171

[97] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999. 33, 36

[98] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):63 –71, summer 1996. 65

[99] Y. Radenac. *Programmation "chimique" d'ordre supérieur*. PhD thesis, Université de Rennes 1, 2007. 22, 23, 49, 65, 158

[100] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, 2004. 37

[101] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM, 2001. 31

[102] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(4):189–193, 1989. 42

[103] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989. 41

[104] M. Raynal. A distributed solution to the k-out of-m resources allocation problem. *Advances in Computing and Information–ICCI'91*, pages 599–609, 1991. 42, 43, 183

[105] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM SIGOPS Operating Systems Review*, 25(2):47–50, 1991. 39

[106] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981. 40, 42, 43

[107] G. Ricart and A. K. Agrawala. Authors' response to "on mutual exclusion in computer networks" by carvalho and roucairol. *Commun. ACM*, 26(2):147–148, February 1983. 41

[108] L. Rilling and C. Morin. A practical transparent data sharing service for the grid. In *IEEE International Symposium on Cluster Computing and the Grid*, volume 2, pages 897–904. IEEE, 2005. 68

[109] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001. 33, 69, 93, 174

[110] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems (TOCS)*, 5(3):284–299, 1987. 40, 183

[111] C. Schmidt and M. Parashar. Squid: Enabling search in dht-based systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, 2008. 38, 94, 129, 174

[112] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22, 1990. 71, 94, 156

[113] H. J. Siegel. Interconnection networks for simd machines. *Computer*, 12(6):57–65, 1979. 35

[114] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley, 1998. 39

[115] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5):651–662, 1989. 41

[116] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3), 1983. 95, 175, 179, 184

[117] P. K. Srimani and R. L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, 1992. 42

[118] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001. 32, 69, 93, 174

[119] H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine. *ACM SIGARCH Computer Architecture News*, pages 105–117, 1977. 20

[120] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985. 41, 43

[121] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979. 40

[122] M. Tsangou, S. Ndiaye, M. Seck, and W. Litwin. Range queries to scalable distributed data structure rp*. In *Proc. Fifth Workshop on Distributed Data and Structures, WDAS*, 2003. 37

[123] M. Viroli and F. Zambonelli. A Biochemical Approach to Adaptive Service Ecosystems. *Information Sciences*, 2009. 171

[124] Y. Wang, M. Li, J. Cao, F. Tang, L. Chen, and L. Cao. An ECA-Rule-Based Workflow Management Approach for Web Services Composition. In *4th International Conference on Grid and Cooperative Computing (GCC 2005)*, pages 143–148, Beijing, China, 2005. 5

[125] K.-L. A. Wong. *Jasmine: A shared-object multi-locking distributed shared memory system for heterogeneous computers*. PhD thesis, University of Queensland Australia, 2004. 65

[126] Z. Xu and Z. Zhang. Building low-maintenance expressways for p2p systems. *Hewlett-Packard Labs, Palo Alto, CA, Tech. Rep. HPL-2002-41*, 2002. 36

[127] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004. 36

List of Publications

International Journals

[BOT13] Marin Bertier, **Marko Obrovac** and Cédric Tedeschi, "Adaptive Atomic Capture of Multiple Molecules", *Journal of Parallel and Distributed Computing (JPDC)*, 2013. To appear.

[OT13] **Marko Obrovac** and Cédric Tedeschi, "Experimental Evaluation of a Hierarchical Chemical Computing Platform", *International Journal of Networking and Computing (IJNC)*, 2013. Special issue on APDCM 2012. To appear.

[OT12] **Marko Obrovac** and Cédric Tedeschi, "Distributed Chemical Computing: A Feasibility Study", *International Journal of Unconventional Computing (IJUC)*, 2012. To appear.

International Conferences

[FOT13] Héctor Fernández, **Marko Obrovac** and Cédric Tedeschi, "Towards Decentralised Workflow Scheduling via a Rule-driven Shared Space", *13th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)*. Short paper. Florence, Italy, 2013. To appear.

[OT13a] **Marko Obrovac** and Cédric Tedeschi, "Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models", *14th International Conference on Distributed Computing and Networking (ICDCN)*, Mumbai, India, 2013.

[BOT12] Marin Bertier, **Marko Obrovac** and Cédric Tedeschi, "A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms", *13th International Conference on Distributed Computing and Networking (ICDCN)*, Hong Kong, China, 2012.

International Workshops

[OT12a] **Marko Obrovac** and Cédric Tedeschi, "On the Feasibility of a Distributed Runtime for the Chemical Programming Model", *14th International Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, Shanghai, China, 2012, held in conjunction with IEEE IPDPS 2012.

[OT12b] **Marko Obrovac** and Cédric Tedeschi, "When Distributed Hash Tables Meet Chemical Programming for Autonomic Computing", *15th International Workshop on Nature Inspired Distributed Computing (NIDisC)*, Shanghai, China, 2012, held in conjunction with IEEE IPDPS 2012.

National Conferences

[BOT13a] Marin Bertier, **Marko Obrovac** and Cédric Tedeschi, "Un protocole pour la capture atomique de molécule", *Conférence Française en Système d'Exploitation*, Grenoble, France, 2013.

Technical Reports

[TR-8145] **Marko Obrovac** and Cédric Tedeschi, "Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models", Technical report N° RR-8145, 2012.

[TR-7925] Héctor Fernández, **Marko Obrovac** and Cédric Tedeschi, "Decentralised Multiple Workflow Scheduling via a Chemically-coordinated Shared Space", Technical report N° RR-7925, 2012.

[TR-7661] **Marko Obrovac** and Cédric Tedeschi, "On Distributing the Runtime of the Chemical Programming Model", Technical report N° RR-7661, 2011.

Appendix A

Résumé en Français

Avec l'adoption croissante des architectures orientées service (SOA), les plates-formes à large échelle ont récemment subi une nouvelle mutation dans leur forme et leur utilisation. Dans ces plates-formes, l'entité de base est un service, c'est à dire, l'encapsulation d'un certain calcul, d'une facilité de stockage ou d'un capteur, lequel peut être utilisé par des utilisateurs, éventuellement en combinaison avec d'autres services. La composition de ces services doit se faire au-dessus d'un environnement qui est hautement distribué et dynamique. Dans ce contexte, où la présence d'un orchestrateur centralisé est difficile à assurer, l'informatique autonome [96], dont l'objectif est de construire des systèmes distribués de calcul affichant des propriétés tels que l'auto-réparation, l'auto-adaptation ou l'auto-optimisation semble un chemin prometteur. Comme il est argumenté dans [96], il est en particulier nécessaire de proposer des nouvelles abstractions de programmation pour exprimer cette autonomie. Les langages déclaratifs [76], et en particulier les langages à base de règles répondent à cette problématique, comme il a été indiqué récemment dans les articles [3, 51].

Les chimies artificielles [36], qui sont des modèles de programmation à base de règles, et dont le modèle d'exécution, simple, est inspirés par l'observation des processus chimiqies, ont bénéficié d'un intérêt nouveau dans ce contexte, et sont désormais utilisés pour modéliser cet *écosystème* de services [123]. Plus concrètement, le modèle de programmation chimique, initialement développé pour écrire des programmes hautement parallèles, a été identifié comme un paradigme qui fournit un bon niveau d'abstraction pour ce contexte. Des exemples concrets montrant cette adéquation peuvent être trouvés dans [15, 41, 89, 34].

Dans la version de base du modèle de programmation chimique [11], un programme est envisagé comme une *solution chimique* dans laquelle des molécules de données flottent et réagissent conformément à certaines *règles de réaction* précisant les actions du programme, afin de produire de nouvelles données (les produits des réactions). Lors de

l'exécution, les réactions surviennent de façon autonome et parallèle, et donc dans un ordre non déterministe. Une fois qu'il ne reste plus de réactions réalisables, la solution contient le résultat du calcul.

Alors que le paradigme chimique facilite l'expression des systèmes autonomes, l'exécution de programmes chimiques sur des plates-formes distribuées est encore un problème largement ouvert. Parmi les obstacles fondamentaux, il y a la capture atomique des molécules satisfaisant une réaction. Lors de l'exécution, une molécule peut potentiellement participer à plusieurs réactions possibles. Toutefois, elle peut, au final, n'être consommée qu'une unique fois. Dans le cas contraire, la logique du programme serait rompue. Ce problème est illustré dans la section A.1.

Raffinons légèrement le problème : nous considérons un programme chimique constitué d'un multi-ensemble de données et d'un ensemble de règles qui agissent simultanément sur ce multi-ensemble. Les données et les règles sont réparties sur un ensemble de nœuds sur lequel le programme s'exécute. Chaque nœud tente régulièrement d'obtenir les molécules (données) nécessaires pour appliquer une règle. Vu que plusieurs molécules peuvent satisfaire le modèle de molécule souhaité et les conditions de plusieurs réactions pouvant être effectuées simultanément par des nœuds distincts, la même molécule peut être demandée par plusieurs nœuds en même temps, ce qui mène inévitablement à des conflits. Il est donc nécessaire de rendre atomique l'obtention de l'ensemble des molécules de données nécessaire à l'application de la règle, problème aussi appelé *capture atomique* dans la littérature chimique.

Bien que notre problème ressemble au problème classique d'allocation des ressources [70], il s'en différencie par plusieurs aspects. Premièrement, les molécules sont interchangeables dans une certaine mesure. Les molécules demandées doivent correspondre à un modèle défini dans la règle de réaction qu'un nœud souhaite exécuter. Autrement dit, on distingue deux processus qui sont 1) trouver des molécules correspondant à un modèle (une tâche réalisée par un *protocole de découverte*), et 2) les obtenir pour effectuer des réactions (une tâche réalisée par un *protocole de capture*).

Deuxièmement, et suivant le point précédent, la plate-forme envisagée est à large échelle, et les molécules sont supprimées après avoir réagies, et de nouvelles sont créées. Ainsi, le protocole de découverte des molécules doit être extensible et dynamique. De plus, le nombre de ressources / molécules (et de réactions possibles) fluctue au fil du temps, ce qui va influencer la conception du protocole de capture. Il faut tenir compte du fait qu'une fois que le détenteur d'une molécule a été trouvé, l'échelle du réseau devient de moindre importance, puisque seuls le demandeur et le détenteur de la molécule sont impliqués dans le protocole de capture.

Enfin, et pour résumer, notre objectif est de définir un protocole pour la capture atomique de molécules multiples, lequel s'adapte à la densité des réactions potentielles dans le système de manière dynamique et efficace.

Contribution. Notre contribution est un protocole distribué combinant deux sous-protocoles inspirés de travaux précédents sur l'allocation de ressources distribuées, et adapté à l'exécution distribuée de programmes *chimiques*. Le premier sous-protocole,

appelé *optimiste*, suppose que le nombre de molécules couramment disponibles et correspondant au modèle et à la condition d'une réaction est élevé, de sorte que peu de conflits pour l'obtention des molécules surviendra, les nœuds ayant de fortes chances d'être en mesure de capturer des ensembles distincts de molécules. Bien que ce protocole est simple et rapide, et a un impact limité en termes de communication, il ne garantit pas la vivacité du système lorsque le nombre de conflits augmente. Le second, appelé *pessimiste*, plus lent et plus coûteux en termes de communication, assure la vivacité en présence d'un nombre arbitraire de conflits. Le passage d'un sous-protocole à l'autre est distribué, et se fonde sur les historiques locaux de capture des molécules. Une preuve de la sûreté et de la vivacité de notre protocole est donnée, et son efficacité est discutée à travers un ensemble de résultats de simulation. Soulignons que ces travaux, à notre connaissance, sont les premiers à s'attaquer à l'exécution distribuée des programmes *chimiques*.

Organisation du Résumé. La section suivante présente le paradigme de programmation chimique de façon plus détaillée, et souligne la nécessité de la capture atomique. De plus, elle décrit le modèle du système utilisé. La section A.2 détaille les sous-protocoles, leur coexistence, et le passage de l'un à l'autre. Les preuves de la sûreté, de la vivacité et de l'équité sont également données pour le protocole dans son ensemble. La section A.3 présente les résultats de la simulation et analyse l'efficacité et le coût du protocole. Les travaux connexes sont présentés dans la section A.4. La section A.5 conclut.

A.1 Préliminaires

Différents systèmes nécessitent des algorithmes différents pour assurer l'atomicité des opérations qui varient en complexité. Cette section décrit le modèle de programmation chimique plus en détail, ainsi que le modèle de système utilisé pour notre algorithme.

A.1.1 Modèle de programmation chimique

Le modèle chimique a été initialement proposé pour l'expression naturelle de traitements parallèles, en retirant toute structuration ou sérialisation artificielle, en se concentrant uniquement sur la logique du problème traité par le programme. Suivant l'analogie chimique, les données sont des molécules flottant dans une solution. Elles sont consommées conformément à un ensemble de règles de réaction qui constituent le programme, et qui produisent de nouvelles molécules, c'est à dire des données qui en résultent. Ces réactions ont lieu de manière implicitement parallèle et autonome, jusqu'à ce qu'il n'y ait plus de réactions possibles, un état appelé *inertie*. Ce modèle a été initialement formalisé par GAMMA [11], dans lequel la solution est un multi-ensemble de molécules, et les réactions sont des règles de réécriture sur ce multi-ensemble. La consommation est le seul changement d'état possible d'une molécule: une fois qu'elle a été consommée, elle disparaît du multi-ensemble complètement, ce qui signifie que les molécules ne peuvent être que créées ou supprimées, jamais mises à jour ni recréées. En guise d'illustration, con-

sidérons le programme chimique suivant, constitué d'une règle appliquée sur un multi-ensemble d'entiers, et qui en extrait la plus grande valeur:

$$\textbf{let } getmax = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x >= y \textbf{ in } \langle\, 9, 4, 6, 8, 1 \,\rangle$$

La règle *getmax* consomme deux molécules entières et en produit une ayant la valeur la plus élevée des deux. Bien que le résultat du calcul soit déterministe, l'ordre de son exécution ne l'est pas. Il n'y a que l'exclusion mutuelle des réactions par la capture atomique des réactifs qui est implicitement assurée dans le modèle. Ainsi, l'une des d'exécutions possibles est la suivante: d'abord, 1 et 9 réagissent ensemble en même temps que 6 et 8. Ces deux réactions produisent 9 et 8, respectivement. Ensuite, 8 réagit avec 4, et enfin, 9 et 8 réagissent, après quoi l'inertie est atteinte.

Dans la version élevée à l'ordre supérieur du modèle de programmation chimique [13], toute entité prenant part au calcul est représentée par une molécule (y compris les règles), ce qui déclenche une très haute expressivité, capable de traiter naturellement une grande variété de problèmes de coordination rencontrés dans les plates-formes à large échelle [15].

A.1.2 Modèle du Système

Nous considérons un système distribué $\mathbb{DS}$ composé de n machines qui communiquent par passage de messages. Ces machines sont interconnectées par des mécanismes P2P, par exemple une table de hachage distribuée (DHT) [109, 118], permettant de nous abstraire de la communication bas-niveau pour se concentrer sur la capture atomique des molécules.

Diffusion des données et des règles. Ici nous supposons que les données et les règles du programme à exécuter ont déjà été envoyées aux nœuds. Notons que n'importe quelle DHT peut être utilisé à cette fin. Même si les données et les règles sont initialement détenues par une application externe, elle peut les communiquer à un nœud dans la DHT. Le nœud qui a reçu le programme disperse les molécules à travers le réseau. Par la suite, nous supposons que chacun règle du programme est présente sur au moins un nœud du système.

Protocole de Découverte. Pour que la réaction se produise, une combinaison appropriée de molécules doit être trouvée. Bien que les détails de cet aspect soient également omis, ils méritent d'être préalablement discutés. Le mécanisme de recherche de base offert par les DHTs permet uniquement la récupération d'un objet en fonction de son identifiant (unique). Toutefois, dans le cas des règles de réactions, cette *correspondance exacte* fournie par la DHT peut se révéler insuffisante. Il est en effet nécessaire de retrouver *une* molécule satisfaisant un motif (par exemple, de type *entier*) et une condition (par exemple, *supérieur à 3*), comme indiqué dans la section A.1.1. Ceci peut être réalisé grâce à des requêtes sur des intervalles, c'est-à-dire des mécanismes pour trouver certaines (au moins une) molécules appartenant à un intervalle donné de valeurs, à condition que les molécules puissent être totalement ordonnées sur certains critères (éventuellement complexes ou multidimensionnels) [111]. Ce type de mécanismes peut être facilement

étendu pour supporter les motifs et conditions comportant plusieurs molécules. Par exemple, lorsque l'on essaye de capturer deux molécules ordonnées de façon spécifique, un *traducteur de règles* — un programme capable de construire la requête —, en se basant sur la règle donnée et la première molécule obtenue, construit la requête à envoyer au service de découverte de molécules. Si des molécules correspondantes sont trouvées, le protocole de capture sera déclenché.

A.2 Protocole Pour la Capture Atomique de Molécules

Le protocole chargé de la capture atomique des molécules est présenté et discuté dans cette section. Le protocole peut être exécuté selon deux modes, chaque mode utilisant un sous-protocole spécifique, *optimiste*, ou *pessimiste*. Le premier sous-protocole, léger et donc plus efficace, est employé tant que le ratio entre le nombre de réactions possibles et le nombre de captures simultanées est grand. Quand ce ratio passe en dessous d'un certain palier, le sous-protocole pessimiste est activé. Le sous-protocole pessimiste est plus lourd en termes de communications, mais il assure la vivacité du système, même lorsqu'un grand nombre de nœuds cherchent à capturer les mêmes molécules.

A.2.1 Sous-protocole Pessimiste

Fondé sur le protocole de validation à trois phases [116], le sous-protocole pessimiste assure qu'au moins un nœud cherchant à exécuter une réaction la réalisera effectivement. L'obtention d'une molécule s'effectue en trois phases, à savoir (1) la *demande*, (2) la *validation* et (3) l'*obtention*. Elle implique au moins deux nœuds: le nœud demandant la molécule, appelé *demandeur*, et le nœud détenant la molécule, appelé *détenteur*. Les algorithmes A.1 et A.2 présentent le code exécuté sur ces deux entités. La Figure A.1 fournit le diagramme temporel de l'obtention d'une molécule. Lorsque des molécules satisfaisant les conditions d'une réaction ont été trouvées (algorithme A.1, ligne 1), la phase de demande peut commencer (ligne 10).

Le demandeur envoie des messages du type *QUERY* de façon asynchrone à tous les détenteurs pour les informer de son intention de capturer ces molécules. Selon leur état local, chaque détenteur évalue indépendamment le message reçu (lignes 1—13, algorithme A.2) et répond avec l'un des messages suivants: *RESP_OK* (si la molécule demandée est disponible), *RESP_REMOVED* (si la molécule demandée n'existe plus) et *RESP_TAKEN* (si la molécule demandée a été promise à un autre nœud). A moins de recevoir uniquement des messages *RESP_OK*, le demandeur abandonne et envoie le message *GIVE_UP* à tous les détenteurs contactés, les informant ainsi qu'il n'a plus l'intention de capturer leur molécule (ligne 14, algorithme A.1).

La phase de validation suit la phase de demande. Lors de cette phase, les demandeurs envoient des messages *COMMITMENT* aux détenteurs des molécules pour lesquels ils ont obtenu la confirmation de leur disponibilité. Chaque détenteur trie les demandes reçues (ligne 14, algorithme A.2) selon une politique de résolution des conflits (décrite plus loin). L'envoi d'un message *RESP_OK* par un détenteur représente l'engagement de ce dernier

à délivrer sa molécule lors de la troisième phase. Aussi, les messages *QUERY* et *COMMITMENT* reçu par la suite pour cette même molécule (et provenant d'autres nœuds) donneront lieu à l'envoi d'une réponse *RESP_TAKEN*. Si un demandeur ne reçoit pas uniquement des messages *RESP_OK* à ses messages *COMMITMENT*, il abandonne l'obtention des molécules.

Enfin, dans la phase d'obtention, le demandeur envoie des messages *FETCH*, à la réception desquels les détenteurs transmettent la molécule promise, dans un message *RESP_MOLECULE*. A partir de ce moment, les détenteurs envoient des messages *RESP_REMOVED* aux nœuds demandant la molécule.

Résolution des conflits. Chaque détenteur décide individuellement à quel demandeur il envoie une molécule. Puisqu'au moins un demandeur doit être capable d'obtenir ses molécules au complet, chaque détenteur applique la même politique de résolution de conflit (lignes 20—27, algorithme A.2). Lorsqu'au moins deux demandeurs sont en conflits pour l'obtention d'une molécule, les détenteurs donnent la priorité au demandeur avec le nombre de réactions réalisées jusque là (information transmise par les demandeurs dans leur message *QUERY*) le plus bas. En cas d'égalité, le demandeur avec l'identifiant le plus petit obtient la molécule.

A.2.2 Sous-protocole Optimiste

Lorsque la *densité* de réactions possibles est suffisamment grande, et donc que la probabilité d'un conflit est faible, la procédure de capture atomique peut être simplifiée en une approche plus optimiste. Le sous-protocole optimiste ne requiert que deux phases, à savoir l'*obtention*, et la *notification*. L'algorithme A.3 décrit ce sous-protocole du point de vue du demandeur, et l'algorithme A.4 le décrit du point de vue du détenteur. La figure A.2 est un diagramme temporel du sous-protocole optimiste.

Les demandeurs demandent les molécules en parallèle. Une fois qu'un nœud a eu la connaissance de molécules adéquates, il peut lancer la phase d'obtention (ligne 1, algorithme A.3) en utilisant des messages *FETCH*, qui seront évalués du côté du détenteur selon le code de l'algorithme A.4.

Si le demandeur acquiert toutes les molécules, la réaction est lancée, et le demandeur envoie des messages *REACTION* à tous les détenteurs des molécules impliquées pour les notifier de leur consommation. Par la suite, tous ces détenteurs enverront un message *RESP_REMOVED* aux demandes pour ces mêmes molécules. Si un demandeur reçoit ce message, il abandonne la réaction, et en informe les détenteurs des autres molécules qu'il cherche à capturer en leur envoyant un message *GIVE_UP*, ce qui permet alors aux détenteurs de donner les molécules concernés à d'autres demandeurs.

Enfin, lorsque la réponse reçue d'un détenteur est *RESP_TAKEN*, le demandeur attend un temps prédéfini et réessaye de récupérer la molécule. Au bout d'un certain nombre d'échecs, le demandeur abandonne cette réaction.

Résolution des conflits. Etant donné que le protocole optimiste sera employé dans des phases *hautement réactives* de l'exécution du programme, il n'y a pas besoin d'une politique stricte pour la résolution des conflits. Le protocole favorise naturellement le deman-

Algorithm A.1: Sous-protocole pessimiste — demandeur.

```
1  on event combination found
2      QueryPhase(combination);
3  on event response received
4      if phase = query then
5          QueryPhaseResp(resp_mol);
6      else if phase = commitment then
7          CommitmentPhaseResp(resp_mol);
8      else if phase = fetch then
9          FetchPhaseResp(resp_mol);
10 begin QueryPhase(combination)
11     phase ⇐ query;
12     foreach molecule in combination do
13         dispatch QUERY(molecule);
14 begin QueryPhaseResp(resp_mol)
15     if resp_mol ≠ RESP_OK then
16         Abandon(combination);
17     else if all responses have arrived then
18         CommitmentPhase(combination);
19 begin CommitmentPhase(combination)
20     phase ⇐ commitment;
21     foreach molecule in combination do
22         dispatch COMMITMENT(molecule);
23 begin CommitmentPhaseResp(resp_mol)
24     if resp_mol ≠ RESP_OK then
25         Abandon(combination);
26     else if all responses have arrived then
27         FetchPhase(combination);
28 begin FetchPhase(combination)
29     phase ⇐ fetch;
30     foreach molecule in combination do
31         dispatch FETCH(molecule);
32 begin FetchPhaseResp(resp_mol)
33     add resp_mol to reaction_args;
34     if all responses have arrived then
35         Reaction(reaction_args);
36 begin Abandon(combination)
37     phase ⇐ none;
38     foreach molecule in combination do
39         dispatch GIVE_UP(molecule);
```

Algorithm A.2: Sous-protocole pessimiste — détenteur.

```
1  on event message received
2      if message = GIVE_UP then
3          remove sender from molecule.list;
4      else if message.molecule does not exist then
5          reply with RESP_REMOVED;
6      else if message = FETCH then
7          clear molecule.list;
8          reply with molecule;
9      else if molecule has a commitment then
10         reply with RESP_TAKEN;
11     else if message = QUERY then
12         add sender to molecule.list;
13         reply with RESP_OK;
14     else if message = COMMITMENT then
15         SortRequesters(molecule);
16         if molecule.locker = sender then
17             reply with RESP_OK;
18         else
19             reply with RESP_TAKEN;
20 begin SortRequesters(molecule)
21     foreach pair of requesters in molecule.list do
22         if req_j.no_r < req_i.no_r then
23             put req_j before req_i;
24             continue;
25         if req_j.id < req_i.id then
26             put req_j before req_i;
27     molecule.locker ⇐ molecule.list (0);
```

deur qui contacte un détenteur en premier. Toutefois, le protocole optimise n'assure pas qu'une réaction aura lieu dans le cas d'un conflit.

Algorithm A.3: Sous-protocole optimiste — demandeur.

```
1  on event combination found
2      foreach molecule in combination do
3          dispatch FETCH(molecule);
4  on event response received
5      if response ≠ RESP_MOLECULE then
6          Abandon(combination);
7          return;
8      add response.molecule to
       reaction_args;
9      if all responses have arrived then
10         NotifyHolders(combination);
11         Reaction(reaction_args);
12 begin NotifyHolders(combination)
13     foreach molecule in combination do
14         dispatch REACTION(molecule);
15 begin Abandon(combination)
16     foreach molecule in combination do
17         dispatch GIVE_UP(molecule);
```

Algorithm A.4: Sous-protocole optimiste — détenteur.

```
1  on event message received
2      if message = GIVE_UP then
3          molecule.state ⇐ free;
4      else if message = REACTION then
5          remove molecule;
6      else if message.molecule does not
       exist then
7          reply with RESP_REMOVED;
8      else if molecule.state = taken then
9          reply with RESP_TAKEN;
10     else
11         molecule.state ⇐ taken;
12         reply with RESP_MOLECULE;
```

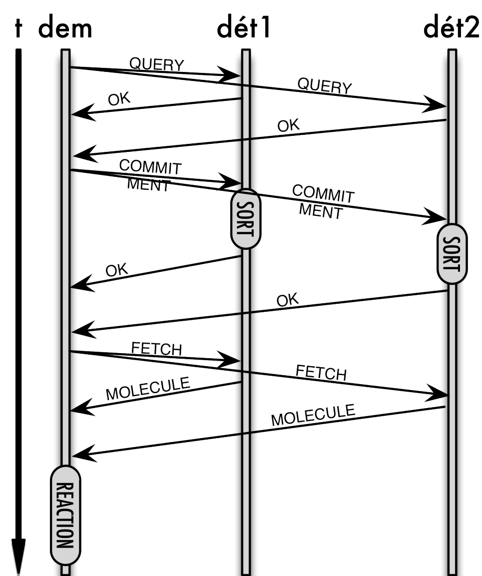


Figure A.1: Échanges pessimistes.

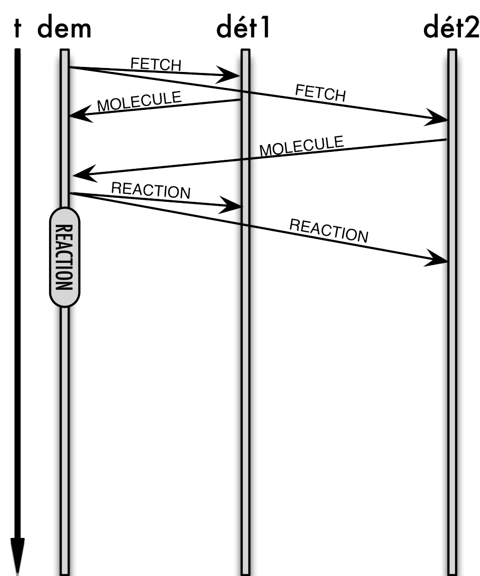


Figure A.2: Échanges optimistes.

A.2.3 Cohabitation des Sous-protocoles

Durant son exécution, un programme passe typiquement par deux stades. Le premier stade est hautement réactif, caractérisé par une grande quantité de réactions possibles. Dans un tel scénario, l'approche optimiste est suffisante pour faire face au faible niveau de concurrence. Le second stade est un stade *silencieux*, avec un nombre de réactions possibles relativement faible. Parce que cela accroit considérablement la probabilité des conflits entre les nœuds, le sous-protocole pessimiste doit alors être employé, pour assurer la vivacité du système. Ainsi, l'environnement d'exécution doit être capable de s'adapter aux changements de stades, et de passer d'un protocole à l'autre en corrélation avec ces changements. De plus, ces protocoles doivent être capable de coexister dans le même environnement, différents nœuds pouvant adopter l'un ou à l'autre de ces protocoles au même moment.

Changement de protocole. Chaque nœud décide indépendamment quel protocole utiliser pour chaque réaction qu'il essaye de réaliser. La décision est d'abord fondée sur le taux de succès local d'un nœud, noté σ_{local}, calculé à partir de l'historique de succès des dernières captures effectuées par le nœud. Un nœud garde aussi la trace des taux de succès d'autres nœuds. Chaque fois qu'un message est envoyé, son expéditeur inclue dans le message son propre taux de succès. On notera σ l'estimation par un nœud du taux de succès global, calculé comme la moyenne arithmétique pondérée du taux de succès local et des taux de succès reçus d'autres nœuds. Enfin, la décision quant au protocole à utiliser dépend de la règle qu'un nœud cherche à exécuter. En effet, ce choix doit dépendre du nombre d'arguments de la règle: plus le nombre de molécules à capturer est grand, plus il sera difficile de les obtenir. Pour capturer r molécules, un nœud utilise le sous-protocole optimiste si et seulement si $\sigma^r \geq s$, où s est une valeur palier prédéfinie. Si cette inégalité n'est pas satisfaite, le nœud utilisera le sous-protocole pessimiste.

Coexistence. De par la localité de la décision de changer de sous-protocole, les différents nœuds du système ne changeront pas à l'exact même moment, entraînant éventuellement des situations où des nœuds essaient d'obtenir les mêmes molécules en utilisant des sous-protocoles différents. Afin de distinguer les demandes optimistes des demandes pessimistes, chaque demandeur inclue un champ *type de demande* dans chaque message envoyé. Sur la base de la valeur de ce champ, le détenteur d'une molécule conflictuelle donnera la priorité aux nœuds utilisant l'algorithme le plus *prudent*, à savoir le pessimiste. Le système assure ainsi qu'un nœud sera capable, *finalement*, d'obtenir toutes les molécules dont il a besoin, puisque le pessimisme est favorisé par rapport à l'optimisme.

A.2.4 Sureté et Vivacité du Système

Le protocole proposé est une combinaison d'extensions de deux protocoles existants, présentés respectivement dans [71] et [116]. Ces deux protocoles ont été initialement introduits pour garantir des transactions sur des ressources qui ont un seul détenteur. Dans notre contexte, un demandeur peut demander plusieurs molécules détenu par différents nœuds.

Ces protocoles doivent garantir deux propriétés.

- **Sûreté**: Chaque molécule utilisée le sera dans une et une seule réaction (en considérant que toutes les réactions consomment toutes les molécules y prenant part).
- **Vivacité**: Si un nœud envoie une demande infiniment souvent, il finira par réussir à capturer les molécules, si ces molécules sont disponibles suffisamment longtemps.

A.2.4.1 Sûreté

Les deux protocoles sur lesquels nous nous appuyons ont cette propriété indépendamment, assurant la sûreté lorsque les sous-protocoles sont utilisés unanimement. Il y a deux cas possibles lorsque les deux sous-protocoles sont utilisés de façon concurrente. Lorsqu'une demande optimiste arrive avant une demande pessimiste, la demande pessimiste est rejetée parce que la molécule a déjà été réservée par le demandeur optimiste. Dans l'autre cas, si une demande pessimiste arrive d'abord, la demande optimiste est rejetée à la faveur de la demande pessimiste.

A.2.4.2 Vivacité

Pour établir la vivacité, nous montrons:

1. que s'il n'y a pas de réaction qui se produit, les nœuds changent de sous-protocole pour le sous-protocole pessimiste;
2. que si plusieurs demandeurs pessimistes sont en conflit, au moins une réaction n'est pas abandonnée;
3. un nœud ne peut pas voir sa réaction infiniment abandonnée.

Dans le cas d'un conflit entre deux demandeurs optimistes, les deux demandes peuvent facilement être abandonnées. Considérons par exemple deux demandeurs concurrents cherchant à capturer deux molécules, A et B. Si le premier demandeur réussit à obtenir A alors que le second réussit à obtenir B, alors les deux demandent vont être abandonnées. Lorsqu'une demande d'un nœud est abandonné, le nœud baisse la valeur de son σ (selon la méthode décrite dans la section A.2.3). S'il y a beaucoup de conflits pendant un certain temps, et d'autant plus s'il n'y a aucune réaction réussie, les valeurs des σ locaux vont décroître pour tous les nœuds, entraînant une situation où les valeurs de σ^r pour les prochaines réactions vont être inférieures au palier s, ce qui va forcer tous les nœuds à utiliser le sous-protocole pessimiste pour lancer de nouvelles demandes.

Pour le sous-protocole pessimiste, nous définissons un ordre total fondé sur le nombre de réactions réussies par un nœud et son identifiant. Dans le cas d'un conflit, toutes les réactions peuvent échouer sauf une — la réaction initiée par le nœud classé premier selon cet ordre total. Parce que l'ordre total est fonction du nombre de réactions réussies, si un nœud, dans le cas d'échecs successifs, essaient infiniment d'obtenir les molécules pour sa

réaction, il réussira finalement, si les molécules sont disponibles suffisamment longtemps, vu qu'il monte dans cet ordre total lorsque d'autres nœuds réussissent à effectuer leurs réactions.

A.3 Evaluation

Nous avons simulé le déroulement de notre protocole afin de mettre en évidence ses performances en fonction de différents scénarios. Le simulateur contruit est à temps discret et a été codé en Python. Il inclut une DHT qui permet la distribution des molécules entre les nœuds du système. Notre algorithme de capture s'appuie sur cette DHT.

Toutes nos expériences reposent sur un réseau de 250 nœuds exécutant un programme chimique contenant 15 000 molécules. Ce programme est basé sur une règle simple qui consomme deux molécules et n'en produit aucune nouvelle. Ce programme simple nous permet de nous concentrer sur l'évaluation du protocole de capture de molécules. Chaque expérience est exécutée 50 fois et les courbes ci-dessous affichent des valeurs obtenues en moyennant les résultats des 50 exécutions.

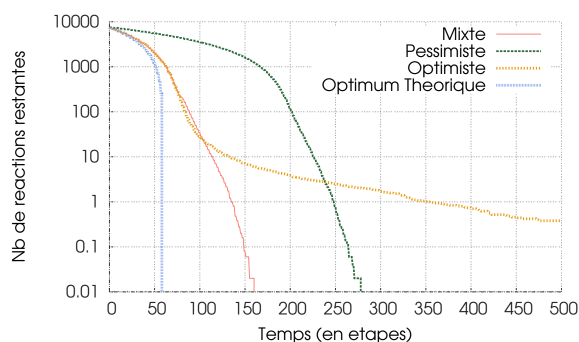


Figure A.3: Comparaison des performances des protocoles.

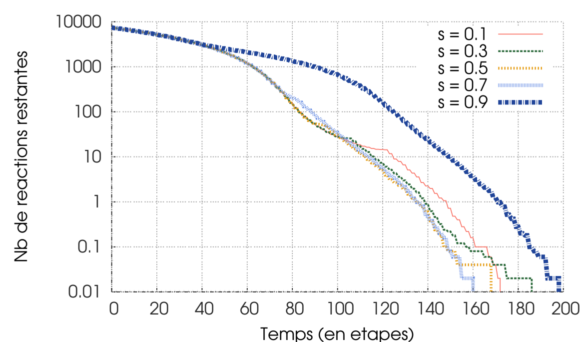


Figure A.4: Temps d'exécution en fonction de la borne.

Expérience 1. Dans un premier temps, nous évaluons séparément les deux sous-protocoles. La figures A.3 montre le nombre de réactions qui restent à exécuter à chaque étape jusqu'à inertie pour trois scénarios :

- l'utilisation exclusive du sous-protocole optimiste,
- l'utilisation exclusive du sous-protocole pessimiste,
- notre protocole complet qui change de mode (avec $\sigma = 0.7$).

Il est important de noter que l'échelle est logarithmique. Cette figure montre qu'en utilisant exclusivement le sous-protocole *optimiste*, le nombre de réactions réalisées est très important au début des expériences, c'est-à-dire, quand beaucoup de molécules sont

disponibles, et donc que les risques de conflits sont faibles. La situation devient plus critique quand il reste peu de réactions à exécuter. En effet, les quelques réactions qui restent à exécuter ne sont jamais réalisées car des conflits se produisent inexorablement, empùłchant leur capture. Quand les nœuds sont tous *pessimistes*, le nombre de réactions qui ont lieu à chaque étape est relativement constant dans le temps. L'inertie est donc atteinte en un temps raisonnable, grùćce à la vivacité qui est garantie dans ce mode de fonctionnement. Dans la plupart des étapes (jusqu'à l'étape 115 environ) le nombre de réactions réalisées dans le protocole *mixte* est le mùłme que pour le protocole *optimiste*, ce qui signifie que les nœuds utilisent majoritairement le protocole *optimiste*. Cependant, à la fin de l'exécution, le système parvient à atteindre rapidement l'inertie en utilisant le protocole *pessimiste*. Après le changement de protocole, la courbe de notre protocole *mixte* diverge de la courbe du protocole *optimiste* pour ressembler à celle du protocole *pessimiste* en affichant un gain de performance de 42%.

Finalement, la courbe *optimum théorique* représente le nombre d'étapes nécessaires pour effectuer l'ensemble des réactions dans un système distribué *idéal*, c'est-à-dire avec aucun conflit et avec un parallélisme optimal, chaque nœud utilisant systématiquement le protocole optimiste. En comparaison, le protocole *mixte* nécessite 166% plus d'étapes à cause des conflits qui se produisent.

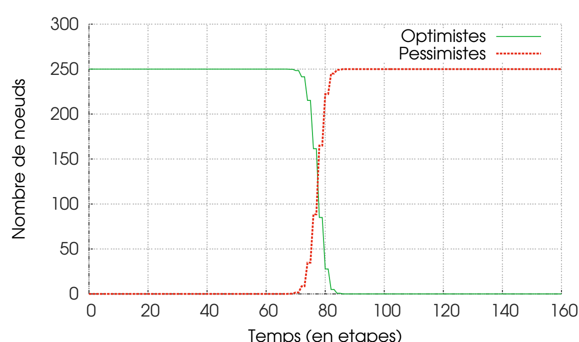


Figure A.5: Nombre de nœuds utilisant les protocoles optimistes et pessimiste.

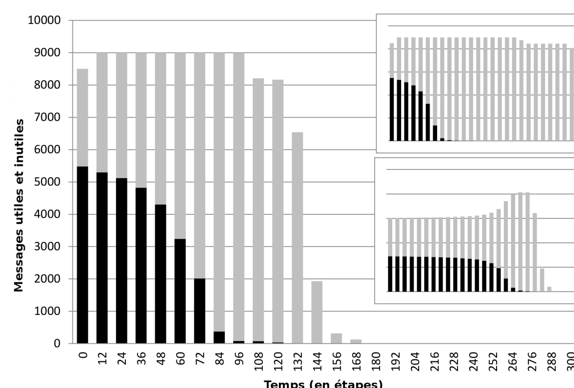


Figure A.6: Nombre de messages échangés par cycle.

Expérience 2. Nous désirons maintenant évaluer l'impact de la borne s (qui décide du moment oùź les nœuds changent de protocole) sur les performances du système. La figure A.4 montre, avec une échelle logarithmique, le nombre de réactions qui reste à exécuter dans le système pour différentes valeurs pour la borne s allant de 0.1 à 0.9. De façon attendue les courbes sont similaires au début de l'exécution car tous les nœuds utilisent le protocole optimiste. La première courbe à diverger est celle oùź la borne vaut 0.9, car les nœuds changent de protocole tùťt, et ne profitent pas suffisamment des performances du protocole *optimiste*. Elle est, donc, la dernière à atteindre l'inertie. Ensuite, les autres courbes divergent à différents moment et donc complètent leur exécution à différents moments. La figure A.4 nous pousse à conclure que, des cinq valeurs testées pour la borne

s, la valeur 0.7 présente les meilleures performances. L'objectif de cette thèse n'est pas d'affiner la valeur de la borne *s*, mais juste de montrer son influence.

Expérience 3. Ici, nous examinons la façon dont les nœuds passent d'un protocole à l'autre. La figure A.5 montre qu'au début de l'exécution, tous les nœuds commencent à collecter des molécules en utilisant le protocole *optimiste*. Le changement de protocole arrive environ à la moitié de l'exécution. A cet instant, les nœuds n'arrivent plus à capturer efficacement des molécules avec le protocole *optimiste*, et, en conséquence, changent de sous-protocole pour le *pessimiste*. Nous observons que, grùćce à l'échange des valeurs σ entre les nœuds, les nœuds changent de sous-protocole rapidement: pour un système de 250 nœuds, seulement 15 étapes sont nécessaires pour que tous les nœuds changent de protocole, ce qui représente 10% du temps d'exécution.

Expérience 4. Finalement, nous étudions les coùżts en termes de communication entre les nœuds du système. La figure A.6 montre le nombre de messages échangés à chaque cycle (chaque cycle correspond à 12 étapes de simulation), répartis en deux catégories : messages utiles (ceux qui permettent de parvenir à la réaction, en noir) et messages inutiles (ceux qui correspondent à des réactions annulées à cause des conflits, en gris). Nous remarquons que le protocole *mixte* permet de prendre les avantages des deux protocoles. Premièrement, il profite du haut pourcentage de messages utiles du protocole *optimiste*. Ensuite, le changement de protocole permet de diminuer le nombre total de messages. Quand nous comparons le coùżt de communication des deux sous-protocoles (optimiste et pessimiste), affiché sur le cùťté droit de la figure A.6, nous voyons que le changement de protocole réduit le trafic réseau et facilite le passage à l'échelle.

A.4 Travaux Connexes

Le paradigme chimique a été à l'origine conçu pour les programmes qui doivent être exécutés sur des machines parallèles. Le travail pionnier de Banâtre *et al.* [11] propose deux approches conceptuelles de la mise en oeuvre, dans lesquelles chaque processeur d'une machine parallèle détient une molécule et la compare avec les molécules de tous les autres processeurs. Une approche légèrement différente a été proposée dans le travail de Linpeng *et al.* [56], où un programme est exécuté en plaçant les molécules sur une bande et en les pliant après chaque comparaison verticale. Récemment, Lin et al. ont développé un analyseur de programmes GAMMA pour les exécuter sur un cluster exploitant la puissance de calcul des GPUs [75]. Tous les travaux mentionnés présentent d'importantes caractéristiques d'accélération d'exécution, mais les plates-formes utilisées sont plutôt limitées.

L'exclusion mutuelle et les algorithmes d'allocation des ressources ont été largement étudiés. Néanmoins, la plupart des travaux se concentre sur le partage d'une ressource spécifique, ou une section critique, parmi de nombreux processus [110, 28]. Une solution de base du problème *k-parmi-M* a été donnée par Raynal [104]. Ce premier travail

est un algorithme statique à base de permission dans lequel seul le numéro d'un ensemble prédéfini de ressources varie du nœud au nœud. En outre, la solution suppose une connaissance globale du système. Par contre, un environnement d'exécution pour les programmes chimiques est un système dynamique dans lequel les nœuds doivent obtenir des molécules distinctes, qui peuvent être considérées comme des ressources.

Le protocole de validation à trois phases a été proposé comme mécanisme de recouvrement de fautes pour les systèmes de bases de données distribuées [116]. Bien que, dans son essence similaire au protocole de validation, l'objectif du sous-protocole optimiste proposé ici est d'assurer la vivacité du système en assurant que au moins un nœud sera en mesure d'achever sa réaction.

A.5 Conclusion

Alors que la métaphore chimique gagne de l'attention dans la modélisation de la coordination autonomes de services, le déploiement effectif des programmes suivant le modèle de programmation chimique sur les plates-formes distribuées reste un problème très ouvert. Dans cette thèse, nous avons décrit un nouveau protocole pour capturer plusieurs molécules de façon atomique dans un multi-ensemble évolutif d'objets distribués au-dessus d'une plate-forme à large échelle. En passant dynamiquement d'un sous-protocole à l'autre, notre protocole exploite pleinement leurs bonnes propriétés (impact limité sur le réseau et rapidité du sous-protocole optimiste lorsque la densité des réactifs est élevée, et garantie de la vivacité du sous-protocole pessimiste une fois que cette densité chute), sans souffrir de leurs inconvénients. Ces caractéristiques sont illustrées à travers un ensemble de simulations.

Ce protocole s'inscrit dans le cadre d'un travail ambitieux qui vise à la construction d'une plate-forme distribuée autonome offrant toutes les fonctionnalités nécessaires pour exécuter des programmes chimiques. Ce travail a aussi l'intérêt de revisiter des problèmes classiques des systèmes distribués, en gardant les exigences de passage à l'échelle et les spécificités du modèle chimique à l'esprit. En suivant cette idée, cet article aborde l'exclusion mutuelle. Dans notre contexte, la propriété de vivacité est d'abord une propriété du système — au moins une réaction doit avoir lieu en un temps fini. Nous montrons aussi l'absence de famine: si des molécules existent suffisamment longtemps, un nœud essayant infiniment souvent de les capturer y parviendra en un temps fini.

Dans le cadre de cette thèse, nous avons élargi ce protocole en proposant un système complet pour l'exécution de programmes *chimiques* sur des plates-formes à large échelle. Plus spécifiquement, cette thèse décrit un modèle d'exécution entièrement décentralisé qui permet d'exécuter des programmes chimiques dans des systèmes hautement dynamiques et hétérogènes en s'appuyant sur le protocole de capture décrit. Ainsi, un prototype a été conçu prouvant la viabilité d'un tel systéme. En outre, il a été testé sur Grid5000 [20], et les expériences conduites affirment ses bonnes propriétés, notamment la faisabilité et la scalabilité. Nous avons l'intention d'utiliser la plate-forme pour exploiter l'expressivité du paradigme chimique et en particulier développer un système de gestion

de workflows tel qu'il a été défini dans [41],

VU:	VU:
Le Directeur de Thèse	Le Responsable de l'École Doctorale
Thierry PRIOL	_____ ______

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication:

Le Président du jury,

Jean-Pierre BANÂTRE